Composition Assessment Metrics for CBSE

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

Abstract

Objective: *Formal definition of composition assessment metrics for CBSE, using an extension of the CORBA Component Model metamodel as the ontology for describing component assemblies.*

Method: *Representation of a component assembly as an instantiation of the extended CORBA Component Model metamodel. The resulting meta-objects 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: *Formal definition of context-dependent metrics that cover the different composition mechanisms provided by the CORBA Component Model and can be used to compare alternative component assemblies; a metamodel extension to capture the topology of component assemblies.*

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

The pressure for reducing software development life cycles and costs has lead to an increasing interest in Component-Based Development (CBD). These time and cost savings result mainly from reusing software components. Szyperski and Pfister defined a component as "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*" [1]. Among other goals, Component-Based Software Engineering (CBSE) aims to develop enabling techniques for efficient component selection and composition. CBSE researchers and practitioners are keen on developing techniques to achieve predictability of system quality attributes based on the quality attributes of their constituent components [2]. In this context, some authors argue that component evaluation becomes a very important task for CBD [3]. Others, such as Wallnau and Stafford argue that evaluation should be performed on assemblies, rather than on individual components [4]. In our opinion the latter is a more sensible choice, considering that we are fundamentally interested in the overall quality of the produced system, rather than on the individual quality of its components. The interaction among components in an assembly is essential to the overall quality of the system. This means that the "best" component for a particular assembly may not necessarily be the overall "best" component available for that kind of functionality. Instead, the available component that maximizes the overall system quality should be chosen. In this assembly focused view, individual component assessment may be performed as part of the component assembly evaluation, but the focus should be on the overall best solution.

Component assembly evaluation can be of two types: qualitative or quantitative. The former can be performed by an expert, if one is available, but is prone to subjectiveness: different experts may produce conflicting evaluations, and the comparison between evaluations performed on different assemblies is harder. In contrast, the latter can be objective and repeatable, through the usage of well-defined metrics, in the scope of a quality model, thus facilitating the comparison of the results of evaluations performed on different assemblies. The realm of software metrics includes proposals for both product and process assessment. In this paper, we are concerned with product metrics, with a focus on metrics that address reusability. On a recent survey, we have identified several recurrent weaknesses with current metrics proposals for CBD, including the lack of an adequate

quality framework to interpret the metrics, the lack of an ontology we can refer to when implementing a metrics collection system, the usage of an inadequate specifying formalism for the metrics and insufficient validation of the metrics sets [5].

We present a collection of component assembly metrics, including the proposals of Hoek *et al.* [6], and of Narasimhan and Hendradjaya [7]. We formalize those metrics with OCL upon an extension of the CORBA Component Model (CCM) metamodel. Our contribution makes explicit our interpretation of the original metrics definition and provides an executable specification for them. This solves both the ontology and the formalism problems identified earlier and facilitates independent validation efforts.

This paper is organized as follows: In section 2 we discuss related work. In section 3, we introduce the CCM and its underlying metamodel. We identify some of its limitations and propose an extension. In section 4 we provide an overview on our metrics formalization technique. In section 5, we formalize metrics for CBSE upon the extended CCM metamodel. In section 6 we present a metrics collection example. 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 [8] and component assemblies [9]. 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 their focus. Several authors made proposals for the evaluation of component interfaces and dependencies [10-12]. These proposals focus on different aspects of component's interfaces and dependencies 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) [7].

The previous metrics proposals follow a component- centric view of component quality evaluation. In other words, they assess components in isolation. Other metrics proposals follow the assembly-centric evaluation approach. Hoek *et al.* proposed metrics to assess service utilization in component assemblies [6]. Narasimhan and Hendradjaya proposed metrics to assess component interaction density (a measure of the complexity of relationships with other components) [7].

Regardless of being component or assembly focused, all the referred metrics 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 [13] we formalized a metrics set for component reusability assessment [12]. We used that formalization to conduct an independent validation experiment on the same metrics set in [14], 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, which was originally proposed in [15], has also been used in other contexts, therefore with different metamodels. In [16], it was used to formalize 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 [17]. More recently, we have successfully applied the same technique with object-relational database schema metrics [18].

3. CORBA Components

3.1. Standard metamodel

The CCM [19-21] 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.

The CCM specification includes a Meta Object Facility–compliant metamodel [22], where the CCM modeling elements are precisely defined. The metamodel includes three packages. 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, which includes the definitions relating to the component lifecycle.

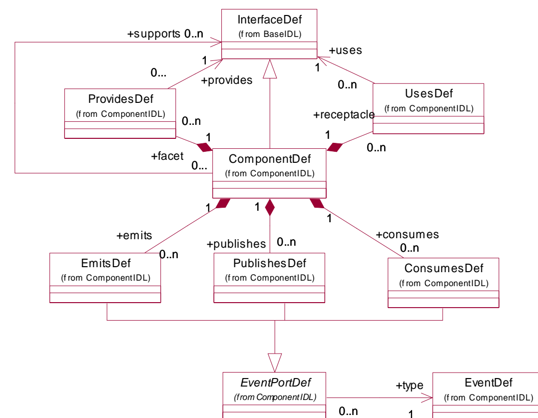

Figure 1 - Excerpt of the CCM metamodel

The CCM has some characteristics that are relevant for the formalization presented in this paper, namely the synchronous and asynchronous interaction mechanisms among components. Each CORBA component may have several provided (facets) and required (receptacles) interfaces, as well as the ability to publish (event sources) and subscribe (event sinks) events. Figure 1 presents an excerpt of the CCM metamodel, relative to the `ComponentIDL` package, where the relationships among these characteristics are presented.

3.2. Metamodel Extension

The CCM metamodel includes abstractions for representing CORBA components, but not for the representation of component assemblies. For instance, it is possible to express that component A has facets X and Y, and that component B has facet Z and receptacle Y, but there is no way of expressing that A's facet Y is wired to B's receptacle Y, with the standard metamodel. This wiring is specified through a Component Assembly Descriptor file. A DTD for such files can be found in [23].

We extended the CCM metamodel to overcome this shortcoming, by adding a new package called Component Assembly Metamodel Extension (`CAME`). In figures 2 through 4, we represent the origin of all meta-classes, except for those defined in the `CAME` package.

Figure 2 represents the extensions required for wiring components through their provided and used interfaces.

These include `ComponentInstanceDef` and `InterfaceConnectorDef` for representing component instances in a component assembly, and a connector between provided and used interfaces ports (`ProvidesPortDef` and `UsesPortDef`).

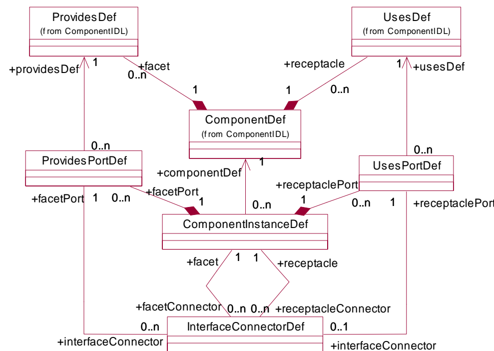

Figure 2 – Metamodel extensions for component wiring through provided and used interfaces

With these new meta-classes, and the associations among them, we have at our disposal the required abstractions for representing component assemblies, including the ports and connectors used for component instances wiring.

Figure 3 includes the meta-classes required for wiring an event emitter with an event consumer. The modeling approach is similar to what was described for provided and required interfaces.

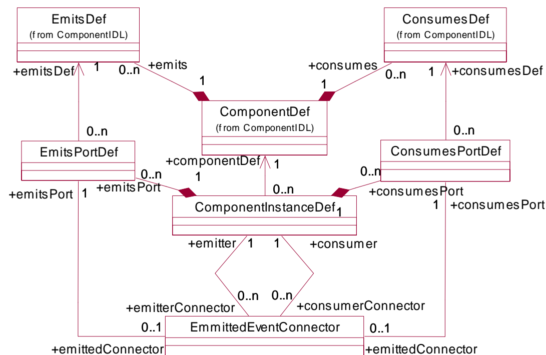

Figure 3 – Metamodel extensions for component wiring through emitted events

Figure 4 includes the meta-classes for representing the wiring between an event broadcaster and the event's consumers.

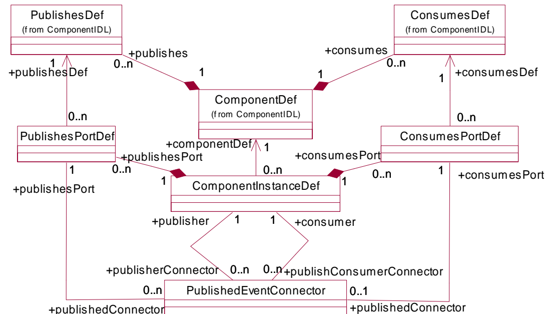

Figure 4 – Metamodel extensions for component wiring through published events

Finally, we need an abstraction to represent the component assembly. A component assembly may have an arbitrary number of component instances (Figure 5).

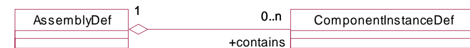

Figure 5 - The component assembly meta-class

4. Metrics definition with OCL

A CCM assembly can be represented as an instance of the extended 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.

OCL expressions can be used to traverse this graph and collect the required information from the meta-data (meta-objects and meta-links) in order to compute the assembly metrics.

In our approach, OCL functions are defined in the context of a meta-class to facilitate information retrieval from instances of that meta-class. Consider the following example, where we define two functions, `EmittedEventTypes` and `EmittedEventTypesCount`, in the context of the **`ComponentInstanceDef`** meta-class, to represent the set of event types that may be emitted by that component, as well and the number of elements in that set, respectively:

```
ComponentInstanceDef

EmittedEventTypes(): Set (EventDef) =
  self.componentDef.emits->collect(type)->
    asSet()

EmittedEventTypesCount(): Integer =
  self.EmittedEventTypes()->size()
```

5. Component assembly metrics definition

In this section, we formalize several metrics for CBD proposed in the literature. 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. Then, we present 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*. The latter may include auxiliary functions. The `AuxilaryFunction` typeface is used to identify these functions. The final expression of the metric is expressed within a frame.

We start by formalizing the proposals of Hoek et al. [6], and of Narasimhan and Hendradjaya [7]. Overall, they aim to assess the fitness of components in a specific architecture. These metrics are context specific, in the sense that the same component will have a different metric value depending on the assembly where it is being integrated.

5.1. Component service utilization metrics [6]

(i) The Provided Services Utilization (*PSU*) represents the ratio of services provided by the component which are actually used (Eq. 1). The Required Services Utilization (RSU) is similar, but for required services (Eq. 2).

$$PSU_X = \frac{P_{actual}}{P_{total}} \quad \text{(Eq. 1)}$$

where:
P_{actual} = number of services provided by component X that are actually used by other components
P_{total} = number of services provided by component X

$$RSU_X = \frac{R_{actual}}{R_{total}} \quad \text{(Eq. 2)}$$

where:
R_{actual} = number of services required by component X that are actually provided by the assembly
R_{total} = number of services required by component X

(ii) *PSU* denotes the extent to which the assembly uses the services provided by the component. Although a low value of PSU may occur if a component was built for reuse, it also means that the component carries a large amount of extra functionality that is not required by the assembly. *RSU* denotes the extent to which a component requires services that are available in the component assembly. Ideally, *RSU*'s value should be 1.

(iii) The notion of service, as presented by Hoek *et al.* is fuzzy, in the sense that it covers any kind of publicly accessible resource of the component, including operations, data structures and so on. The granularity of what is considered a service also varies, according to which is more suitable for the Architecture Description Language expressing the component assembly we want to analyze. In our formalization we assume interfaces as services that a component can provide, or require, as this is inline with the granularity level established in the CCM for component dependency. The **`PSU`** and **`RSU`** functions formalize these metrics in OCL.

```
ComponentInstanceDef

ProvidedInterfacesUsed(): Set(InterfaceDef) =
  self.facetConnector.facetPort.providesDef.
    provides->asSet()

PActual(): Integer =
  self.ProvidedInterfacesUsed()->size()

ProvidedInterfaces(): Set(InterfaceDef) =
  self.componentDef.facet.provides->asSet()

PTotal(): Integer =
  self.ProvidedInterfaces()->size()
```

```
PSU(): Real = self.PActual()/self.PTotal()
```

```
RequiredInterfacesAvailable():
  Set(InterfaceDef) =
    self.receptacleConnector.receptaclePort.
      usesDef.uses->asSet()
```

```
RActual(): Integer =
  self.RequiredInterfacesAvailable()->size()

RequiredInterfaces(): Set(InterfaceDef) =
  self.componentDef.receptacle.uses->asSet()

RTotal(): Integer =
  self.RequiredInterfaces()->size()
```

```
RSU(): Real = self.RActual()/self.RTotal()
```

5.2. Compound Service Utilization metrics [6]

(i) The Compound Provided Service Utilization (*CPSU*) represents the ratio of services provided by the components in the assembly which are actually used (Eq. 3). The Compound Required Service Utilization (*CRSU*) is similar, but for services required by the components (Eq. 4).

$$CPSU = \frac{\sum_{i=1}^{n} P^{i}_{actual}}{\sum_{i=1}^{n} P^{i}_{total}} \quad \text{(Eq. 3)}$$

where:
P^{i}_{actual} = number of services provided by component i that are actually used by other components
P^{i}_{total} = number of services provided by component i

$$CRSU = \frac{\sum_{i=1}^{n} R^{i}_{actual}}{\sum_{i=1}^{n} R^{i}_{total}} \quad \text{(Eq. 4)}$$

where:
R^{i}_{actual} = number of services required by component i that are actually provided by the assembly
R^{i}_{total} = number of services required by component i

(ii) Low values of *CPSU* may indicate an unbalanced architecture, particularly when combined with high values of *CRSU*, as this implies that the assembly is larger than needed. A low value for *CRSU* suggests that the assembly is not self-sufficient, as more components would are required to provide the remaining functionality.
(iii) The assumptions in this formalization follow those of *PSU* and *RSU*. The `CPSU` and `CRSU` functions are defined in the scope of `Assembly`.

```
Assembly

Components(): Set(ComponentInstanceDef) =
  self.contents->select(c |
    c.oclIsKindOf(ComponentInstanceDef))->
      asSet()

SUM_PActual(): Integer =
  self.Components()->collect(PActual())->sum()

SUM_PTotal(): Integer =
  self.Components()->collect(PTotal())->sum()
```

```
CPSU(): Real = self.SUM_PActual() /
               self.SUM_PTotal()
```

```
SUM_RActual(): Integer =
  self.Components()->collect(RActual())->sum()

SUM_RTotal(): Integer =
   self.Components()->collect(RTotal())->sum()
```

```
CRSU(): Real = self.SUM_RActual() /
               self.SUM_RTotal()
```

5.3. Interaction density of a component [7]

(i) The Interaction Density of a Component (*IDC*) is defined as a ratio of actual interactions over potential ones (Eq. 5). The Incoming and Outgoing Interaction Density of a Component (*IIDC* and *OIDC*, respectively) are similar, but considering only incoming interactions (Eq. 6) or outgoing ones (Eq. 7).

$$IDC = \frac{\#I}{\#I_{\max}} \quad \text{(Eq. 5)}$$

where
$\#I$ = Actual Interactions
$\#I_{\max IN}$ = Maximum available interactions

$$IIDC = \frac{\#I_{IN}}{\#I_{\max IN}} \quad \text{(Eq. 6)}$$

where
$\#I_{IN}$ = Actual incoming interactions
$\#I_{\max IN}$ = Maximum available incoming interactions

$$OIDC = \frac{\#I_{OUT}}{\#I_{\max OUT}} \quad \text{(Eq. 7)}$$

Where
$\#I_{OUT}$ = Actual outgoing interactions
$\#I_{\max OUT}$ = Maximum available outgoing interactions

(ii) A higher interaction density causes a higher complexity in the interaction. Narashiman and Hendrajaya regard this complexity as a source of risk that should be taken into account when assigning professionals to component design. The rationale would be to assign the most experienced developers to denser interactions.
(iii) *IDC* is somewhat similar to a combination of *PSU* with *RSU* but Narasimhan and Hendradjaya consider the usage of both interfaces and events as interactions. The parallel can also be made for *IIDC* with *RSU*, and *OIDC* with *PSU*. `IDC`, `IIDC` and `OIDC` can be formalized in OCL, as follows:

```
ComponentInstanceDef

EmittedEventsConsumed(): Set(EmitsDef) =
  self.emitterConnector.emitsPort.
    emitsDef->asSet()

EmittedEventsConsumedCount(): Integer =
  self.EmittedEventsConsumed()->size()

EmittedEvents(): Set (EmitsDef)=
  self.componentDef.emits

EmittedEventsCount(): Integer =
  self.EmittedEvents()->size()

PublishedEventsConsumed(): Set (PublishesDef)=
  self.publisherConnector.publishesPort.
    publishesDef->asSet()

PublishedEventsConsumedCount(): Integer =
  self.PublishedEventsConsumed()->size()

PublishedEvents(): Set (PublishesDef)=
  self.componentDef.publishes

PublishedEventsCount(): Integer =
  self.PublishedEvents()->size()

FedConsumedEvents(): Set(ConsumesDef) =
  self.emitConsumerConnector.consumesPort.
    consumesDef->union(
      self.publishConsumerConnector.
      consumesPort.consumesDef)->asSet()

FedConsumedEventsCount(): Integer =
  self.FedConsumedEvents()->size()

ConsumedEvents(): Set(ConsumesDef) =
  self.componentDef.consumes

ConsumedEventsCount(): Integer =
  self.ConsumedEvents()->size()

I(): Integer =
  self.FedConsumedEventsCount() +
  self.PublishedEventsConsumedCount() +
  self.EmittedEventsConsumedCount() +
  self.PActual() + self.RActual()

IMax(): Integer =
  self.ConsumedEventsCount() +
  self.PublishedEventsCount() +
  self.EmittedEventsCount() + self.PTotal() +
  self.RTotal()
```

```
IDC(): Real = self.I()/self.IMax()
```

```
IIn(): Integer =
  self.FedConsumedEventsCount() +
  self.RActual()

IMaxIn(): Integer =
  self.ConsumedEventsCount() + self.RTotal()
```

```
IIDC(): Real = self.IIn()/self.IMaxIn()
```

```
IOut(): Integer =
  self.PublishedEventsConsumedCount() +
  self.EmittedEventsConsumedCount() +
  self.PActual()

IMaxOut(): Integer =
  self.PublishedEventsCount() +
  self.EmittedEventsCount() + self.PTotal()
```

```
OIDC(): Real = self.IOut()/self.IMaxOut()
```

5.4. Average Interaction Density of Software Components [7]

(i) The Average Interaction Density of Software Components (*AIDC*) represents the sum of *IDC* for each component divided by the number of components.

$$AIDC = \frac{IDC_1 + IDC_2 + ... + IDC_n}{\#components} \qquad \text{(Eq. 8)}$$

where
IDC_i = *IDC* of component *i*
#components = number of components in the system

(ii) Narashiman and Hendrajaya suggest that *AIDC* can be used as an explanatory variable for assembly complexity, so that a lower value may indicate that less effort can be committed to software risk analysis.
(iii) *AIDC* is somewhat similar to a combination of *CPSU* with *CRSU*. See discussion for *IDC*, *IIDC* and *OIDC*.

```
Assembly

SUM_IDC(): Integer =
   self.Components()->collect(IDC())->sum()
```

```
AIDC(): Real =
   self.SUM_IDC()/self.Components()->size()
```

6. A Metrics collection example

6.1. The car display example

Consider a car company that is specifying the information display system for several car models. The company plans to use a wide range of features from the basic speed, rotations and temperature indicators to driving cameras, a GPS system, and so on. Figures 6 through 8 present the configurations used, from the lower-end model, to the higher-end one.

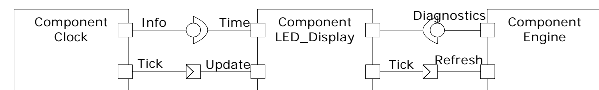

Figure 6 - Low-end car model configuration (model A)

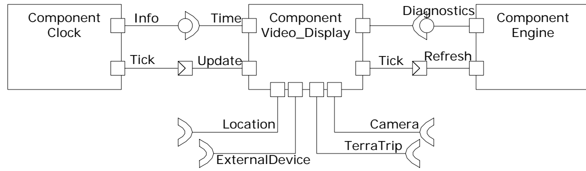

Figure 7 – Middle-range car model configuration (model B)

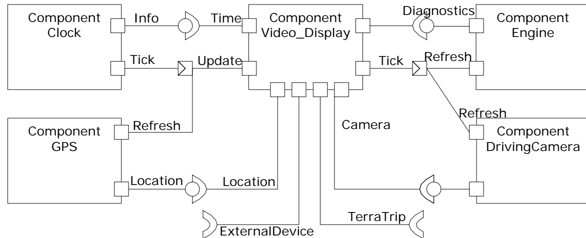

Figure 8 - High-end car model configuration (model C)

6.2. Collected metrics

Table 1 presents the values for the metrics collected on each component instance. The first column presents the component name and the assembly where the metric value was measured – A, B, or C – to identify the corresponding model. NA represents the values which could not be computed (divisions by 0).

Table 1 – Component level metrics

Component	PSU	RSU	IDC	IIDC	OIDC
Clock (A)	1,00	NA	1,00	NA	1,00
LED_Display (A)	NA	1,00	1,00	1,00	1,00
Engine (A)	1,00	NA	1,00	1,00	1,00
Clock (B)	1,00	NA	1,00	NA	1,00
Engine (B)	1,00	NA	1,00	1,00	1,00
Video_Display (B)	NA	0,33	0,50	0,43	1,00
Clock (C)	1,00	NA	1,00	NA	1,00
Engine (C)	1,00	NA	1,00	1,00	1,00
Video_Display (C)	NA	0,67	0,78	0,71	1,00
GPS (C)	1,00	NA	1,00	1,00	1,00
DrivingCamera (C)	1,00	NA	1,00	1,00	1,00

Table 2 presents the assembly level metrics, for each of the proposed car models (A, B, and C).

Table 2 - Assembly level metrics

Assembly	CPSU	CRSU	AIDC
A	1,00	1,00	1,00
B	1,00	0,33	0,83
C	1,00	0,67	0,96

6.3. Discussion

6.3.1. Metrics values The values presented in the previous section illustrate some of the ideas that lead to the proposal of the corresponding metrics. In model A there is a perfect match between the interacting components. All the provided services and emitted events are used or consumed by components within the assembly. In the component assembly of model B, the `Video_Display` component has several required interfaces which are not available within the assembly. In this case, we can suppose that this "waste" of resources occurs mainly because this component was built for reuse. Without additional information, we would prefer model A to model B, if our priority is to have a simpler component that is still able to fulfill our requirements. On the other hand, if we plan to reuse the assembly by adding other components, model B could be a good option, when compared to model A. In terms of effectiveness of reuse, model C is better than B. The comparison of the `RSU` values for the display components can be used to help choosing among different alternatives.

6.3.2. Comments on the metrics' definitions We deliberately kept the original formulae to demonstrate the variability of notations that are commonly used in metrics definitions. A "side-effect" of these informal approaches is that the metrics behavior is sometimes not specified in extreme cases. The original definitions are not clear, with respect to handling the situations marked as NA in Table 1. Another interesting feature of these metrics sets is that they are a lot closer than what one may infer from the presented rationales at a first glance. A higher the service utilization (better used resources), usually corresponds to a higher interaction density (higher complexity). This illustrates potentially overlapping quality attributes that these metrics aim to assess.

All the formalized metrics are defined as ratios where the nominator corresponds to the effective usage of a given mechanism, while the denominator has the maximum possible utilization of the mechanism within the component assembly. This indicates a concern from the metrics proponents to make them dimensionless. This prevents the metrics values from being correlated to the size of the assembly, or the number of times a particular mechanism is used.

7. Conclusions and further work

We have presented a metrics formalization technique that combines the usage of an ontology with a formal and standard specification language. By using an ontology, we were able to unambiguously specify the basic concepts and their relationships used in the metrics definitions. By using OCL, we were able to formally define the metrics, without the burden of complexity that is normally associated with formal languages. This combination of formality with simplicity is essential, considering that metrics can

only be widely adopted if their collection process is replicable on different settings, by different practitioners.

Our contribution to overcome metrics collection replicability is twofold: we introduce a generic metrics specification technique and provide a formal specification for these metrics. A similar approach can be performed with other component metamodels, and for other metrics, as referred in section 2.

In the process of formalizing these metrics, we have detected a limitation in the CCM metamodel: it does not include abstractions for representing component assemblies. We solved this limitation by extending the CCM metamodel for representing component instances and the wiring among them.

References

[1] Szyperski, C. and Pfister, C., "Workshop on Component-Oriented Programming, Summary", in *Special Issues in Object-Oriented Programming - ECOOP'96 Workshop Reader*, M. Mühlhäuser, Ed. Heidelberg: Dpunkt Verlag, 1997.

[2] 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, 2004.

[3] Brownsword, L., Oberndorf, T., and Sledge, C. A., "Developing New Processes for COTS-Based Systems", *IEEE Software*, pp. 48-55, 2000.

[4] Wallnau, K. and Stafford, J. A., "Dispelling the Myth of Component Evaluation", in *Building Reliable Component-Based Software Systems*, I. Crnkovic and Larsson, Eds. Boston, London: Artech House, 2002, pp. 157-177.

[5] Goulão, M. and Abreu, F. B., "Software Components Evaluation: an Overview", *5ª Conferência da APSI*, Lisbon, 2004.

[6] Hoek, A. v. d., Dincel, E., and Medvidovic, N., "Using Service Utilization Metrics to Assess and Improve Product Line Architectures", *9th IEEE International Software Metrics Symposium (Metrics'2003)*, Sydney, Australia, 2003.

[7] Narasimhan, V. L. and Hendradjaya, B., "A New Suite of Metrics for the Integration of Software Components", *The First International Workshop on Object Systems and Software Architectures (WOSSA'2004)*, South Australia, Australia, 2004.

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

[9] Sedigh-Ali, S., Ghafoor, A., and Paul, R. A., "Software Engineering Metrics for COTS-Based Systems", *IEEE Computer*, 2001.

[10] Boxall, M. A. S. and Araban, S., "Interface Metrics for Reusability Analysis of Components", *Australian Software Engineering Conference (ASWEC'2004)*, Melbourne, Australia, 2004.

[11] Gill, N. S. and Grover, P. S., "Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Software", *Software Engineering Notes*, vol. 29, 2004.

[12] Washizaki, H., Yamamoto, H., and Fukazawa, Y., "A Metrics Suite for Measuring Reusability of Software Components", *9th IEEE International Software Metrics Symposium (METRICS 2003)*, Sydney, Australia, 2003.

[13] Goulão, M. and Abreu, F. B., "Formalizing Metrics for COTS", *International Workshop on Models and Processess for the Evaluation of COTS Components (MPEC 2004) at ICSE 2004*, Edimburgh, Scotland, 2004.

[14] Goulão, M. and Abreu, F. B., "Cross-Validation of a Component Metrics Suite", *IX Jornadas de Ingeniería del Software y Bases de Datos*, Málaga, Spain, 2004.

[15] Abreu, F. B., "Using OCL to formalize object oriented metrics definitions", INESC, Software Engineering Group, Technical Report, ES007/2001, May 2001.

[16] Baroni, A. L. and Abreu, F. B., "Formalizing Object-Oriented Design Metrics upon the UML Meta-Model", *Brazilian Symposium on Software Engineering*, Gramado - RS, Brazil, 2002.

[17] Baroni, A. L. and Abreu, F. B., "A Formal Library for Aiding Metrics Extraction", *International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*, Darmstadt, Germany, 2003.

[18] Baroni, A. L., Calero, C., Ruiz, F., and Abreu, F. B., "Formalizing Object-Relational Structural Metrics", *5ª Conferência da APSI*, Lisbon, 2004.

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

[20] Wang, N., Schmidt, D. C., and O'Ryan, C., "Overview of the CORBA Component Model", in *Component-Based Software Engineering: Putting the Pieces Together*, G. T. Heineman and W. T. Councill, Eds., 1 ed: Addison-Wesley Publishing Company, 2001, pp. 557-571.

[21] Estublier, J. and Favre, J.-M., "Component Models and Technology", in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, Eds., 1 ed: Artech House, 2002, pp. 57-86.

[22] OMG, "Meta Object Facility (MOF) Specification (Version 1.4)", Object Management Group, April 2002.

[23] Merle, P., "DTD for Component Assembly Descriptor defined by the CORBA Components Specification 3.0". available online at http://openccm.objectweb.org/dtd/ccm/componentassembly.dtd, last accessed on 21-02-2005., 2003.