A Unifying Framework for the Definition of Syntactic Measures over Conceptual Schema Diagrams (extended version)

Dolors Costal, Xavier Franch

Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona 1-3, Barcelona E-08034, Spain
{dolors|franch}@essi.upc.edu

Abstract. There are many approaches that propose the use of measures for assessing the quality of conceptual schemas. Many of these measures focus purely on the syntactic aspects of the conceptual schema diagrams, e.g. their size, their shape, etc. Similarities among different measures may be found both at the intra-model level (i.e., several measures over the same type of diagram are defined following the same layout) and at the inter-model level (i.e., measures over different types of diagrams are similar considering an appropriate metaschema correspondence). In this paper we analyse these similarities for a particular family of diagrams used in conceptual modelling, those that can be ultimately seen as a combination of nodes and edges of different types. We propose a unifying measuring framework for this family and illustrate its application on a particular type, namely business process diagrams.

Keywords: conceptual schema measure, conceptual schema diagram, metamodelling, MOF.

1 Introduction

Measuring is a fundamental activity for assessing the quality of deployed information systems ("you can't control what you can't measure" [1]). Although most of the existing proposals on software measurement formulate measures for the final software product (e.g., measurement of system performance, reliability, etc.), there is also an important amount of work done on measuring conceptual schemas of the system. These *conceptual schema measures* act as estimators in the earliest phases of software development and may help to detect defects in a cost-effective manner.

Jorgensen and Shepperd reported [2] that in despite of the fact that formal estimation models have existed for many years, the dominant estimation method is still based on expert judgment, which makes measure evaluation subjective and time-consuming and hampers measure reuse. A usual way to minimize expert judgement is to formulate measures based on the structure of conceptual schemas.

There are many approaches that follow this idea to propose syntactic measures over different types of conceptual schema diagrams, such as Entity-Relationship

Diagrams [3], Business Processes [4], Class Diagrams [5], Activity Diagrams [6], Use Cases [7], Workflow Diagrams [8], and Goal-Oriented Diagrams [9]. These measures present some similarities both at the intra-model level (i.e., several measures over the same type of diagram are defined following the same layout) and at the inter-model level (i.e., measures over different types of diagrams are similar considering an appropriate metaschema correspondence).

An analysis of these existing proposals shows that there is a lack of a reference framework for formulating the measures, a lack of guidelines for defining them, and a lack of support for porting them from one kind of diagram to another in spite of these similarities. Our work addresses these issues. To do so, we formulate an approach at the metamodel level such that general-purpose measures can be defined by means of OCL expressions and then we show how they can be specialized and adapted into particular measures for the different types of diagrams mentioned above.

The remainder of the paper is organized as follows. In Section 2 we present an overview of some existing suites of conceptual schema measures. In Section 3 we present our metamodelling framework. In Section 4 we provide an overview of the intended structure of a catalogue of general-purpose measures. In Section 5 we illustrate with an example the use of this catalogue for defining a particular suite of measures for business process diagrams. Finally, in Section 6 we present the validation of our work and in Section 7 we present the conclusions and future work.

2 Background

Measures are applied over different types of conceptual schema diagrams for evaluating different quality attributes. For instance, measures over class diagrams are likely to focus on aspects like maintainability, understandability, etc., whilst measures over business process diagrams may address concepts like liveliness and throughput. The definition, reuse, comparison and validation of these measures has been recognized as a challenge by others (e.g., [10]) triggering then some research that we try to summarize below.

We explore here proposals of measures for: entity-relationship diagrams; class diagrams; business process management diagrams; statechart diagrams; and *i** diagrams. The purpose of this section is not to provide a comprehensive state of the art, that would require a paper by itself, but to show the typical structure-based measures that are defined in these formalisms. Table 1 summarizes some representative measures from the existing proposals.

- **Measures over E-R diagrams.** They triggered the definition of conceptual schema measures. Some of them are present in virtually all proposals of measure suites, like counting number of entities, computing some ratios, etc. Works by Moody [11] and Si-Said Cherif et al. [3] provided a quite comprehensive set of measures that were designed to assess qualities like complexity, analysability and simplicity (i.e., the measures act as indicators of these high-level schema properties).
- **Measures over class diagrams.** Class diagram measures are an evolution of measures on E-R diagrams, as proposed for instance in [3]. Chidamber and Kemerer [5] offered a quite comprehensive measure suite, and Genero et al.

proposed also others related to maintainability and more interestingly, a comprehensive survey including information about their validation [12].

Table 1. Overview of some conceptual schema measures.

Diagram	Measure	Property Measured	Ref.
E-R Diagrams	Number of Entities (E)	Simplicity	[11]
	Number of Entities and Relationships (E+R)	Simplicity	[11]
Class Diagrams	Depth of Inheritance Tree (DIT)	Complexity (behaviour)	[5]
	Number of Children (NOC)	Reusability	[5]
	Number of Associations (NAss)	Maintainability	[12]
Business Processes	Activity Automation Factor (AAF)	Performance	[4]
	Branching Automation Factor (BAT)	Performance	[4]
	Coupling (Coup)	Maintainability	[8]
Statechart Diagrams	Number of Activities (NA)	Maintainability	[13]
	Number of Transitions (NT)	Maintainability	[13]
***i** Diagrams**	Number of SD elements	Complexity	[14]
	Predictability	Accuracy	[15]

- **Measures over business process diagrams.** Business process diagrams are a totally different type of diagram than the two previous ones structure-wise. Among the existing proposals, we mention: a set of 8 structural measures for goal based business process design and evaluation [4]; and coupling and cohesion over workflow models [8].
- **Statechart diagrams.** Genero et al. [13] propose and validate five measures to assess maintainability of UML statechart diagrams that count numbers of states, transitions and their relationships.
- **Measures on *i** diagrams.** There are a few proposals of measures over *i** models. Among them we remark the ones defined in the REACT method [14] which count the different elements of Strategic Dependency (SD) models for obtaining different values. The work in [15] evaluates Strategic Rationale (SR) models by analysing the structure of their means-end and task-decompositions.

A minor point is worth to be remarked. We are using the term “measure” instead of “metric” that for instance is used in a great deal of the papers cited in this section. We have followed the advice by N. Habra et al. [16] among others, that recommend to avoid the use of the term “metrics”: “Though they are widely used in software engineering, we believe that their use causes ambiguity and possibly confusion by suggesting erroneous analogies, e.g. with the mathematical metric in topology, with the metric system of units, etc.”.

In the graph theory area [17], many measures over graphs have been defined. In this work, we will focus on graph measures which evaluate properties that are relevant to assess the quality of conceptual schema diagrams.

3 A Metamodelling Approach for Measure Definition

From the analysis of related work, we observe that conceptual schema measures are all based on the application of a numerical function (e.g., counting or weighting) on the elements that form the language used to create the diagrams under measurement.

For instance, measures over UML class diagrams are based on the number of associations, the number of attributes, etc., and combinations of them. Therefore, we aim at defining a metamodelling approach able to cope with this similarity by unifying the different language metaschemas into one for measurement purposes.

We may observe that the different kind of diagrams targeted in this paper may be reduced to a similar syntactic structure: they are all like graphs such that they differ in their types of nodes and links. Therefore, we make the decision of defining a semantically agnostic metaschema that just reflects this syntactic structure. We adopt as starting point the metaschema for gap typology definition as proposed by Rolland et al. [18] that we modify for adapting it to our needs.

Fig. 1 shows the metaschema. An *Element* is classified according to two different criteria. First, a distinction between *Simple* Element and *Compound* Element is made. Second, an element is classified as a *Node* or an *Edge*. A *Compound* Element is decomposed into finer-grained elements, which can be *Simple* or again *Compound*. *Elements* have one (or eventually more) *category* and optionally a *name*. *Edge* elements are connectors between pairs of elements. One of the connected elements plays the role of the *source* and the other is the *target*. *Edges* may have an *order* to indicate the possible ordering among edges from the same source. It is important to remark that an edge may involve some other edge as source or target, which is quite convenient for being applicable in some contexts. There is a designated compound element that represents a whole *Diagram*. Finally, an element may have associated one or more *Property* and assign a *value* to it. Since the final metaschema has significant differences with Rolland's original one, we name it differently, concretely we called it *Graph-like Metaschema*, or *GLMS* for short. The measures defined over this metaschema will be named *GLMS-measures*.

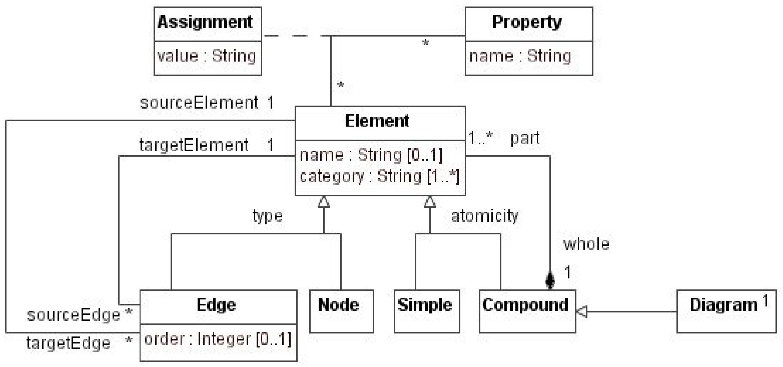


Fig. 1. Graph-like metaschema, adapted from [18].

Table 2 shows the mapping of some concepts from some of the conceptual modelling languages mentioned in the previous section, to GLMS' concepts, with focus on nodes and edges.

Table 2. Correspondence of the GLMS into the concepts of several modelling languages

Diagram	Node	Edge	Property
E-R diagram	Entity, Attribute	Relationship	Multiplicity?
Class diagram	Class, Attribute	Inheritance, Association	Which attribute is key?
Business process diagram	Task, Document	Precedence, Owner	Task executed by human?
*i** diagram	Actor, Dependum	Means-end, Dependency	Committed dependency?

Fig. 2 shows the metamodelling framework that we are proposing. At the M2 level of the four-level metamodel hierarchy [19] we have the several metaschemas for the different conceptual modelling languages: UML diagrams, E-R diagrams, business process modelling formalisms like BPMN, etc. But also the GLMS itself needs to be placed at M2, according to the metamodelling hierarchy classification criteria. Thus, the correspondences established in Table 2 are in fact sub-typing relationships (e.g., E-R diagram is subtype of Diagram). Then, GLMS-measures may be defined (through OCL). They are inherited in the modelling languages metaschemas, and then can be combined as needed to define the measures that apply to this particular language.

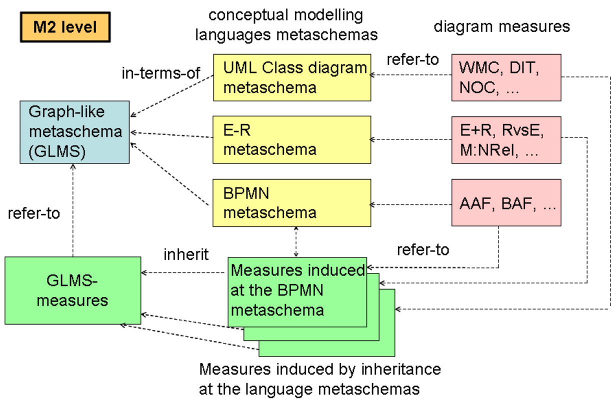

M2 level
conceptual modelling languages metaschemas
diagram measures
in-terms-of
UML Class diagram metaschema
refer-to
WMC, DIT, NOC, ...
Graph-like metaschema (GLMS)
E-R metaschema
E+R, RvsE, M:NRel, ...
BPMN metaschema
AAF, BAF, ...
refer-to
inherit
refer-to
GLMS-measures
Measures induced at the BPMN metaschema
Measures induced by inheritance at the language metaschemas

Fig. 2. Defining model measures in a metamodeling-based framework.

To give an overview let's consider the process of definition of one of the simplest measures, the Number of Entities and Relationships (E+R) measure on ER diagrams:

- At the GLMS we can define a GLMS-measure that counts the number of occurrences of a particular category of element *cat* in a diagram:

```
context Diagram::byCategoryCountDiagram(cat: String): Integer
```

- The E-R metaschema is coupled with the GLMS. In particular, the metaclass *Entity* is defined as subclass of *Node* whilst *Relationship* is defined as subclass of *Edge*. Also, the metaclass *E-RDiagram* is defined as subclass of *Diagram*.
- As a consequence of this subtyping , a measure is induced by inheritance:

```
context E-RDiagram::byCategoryCountDiagram(cat: String): Integer
```

- The *E+R* measure may be defined on top of this inherited measure as:

```
context E-RDiagram::E+R(): Integer
post result = self.byCategoryCountDiagram(“Entity”) +
              self.byCategoryCountDiagram(“Relationship”)
```

More details are rendered in the next sections. In particular:

- Which measures need to be defined over the GLMS?
- How these measures can be inherited over modelling languages metaschemas?

4 Defining Measures over the Graph-like Metaschema

To make our approach usable, we need to define a comprehensive catalogue of GLMS-measures. It is not a goal of this paper to produce such a catalogue. However we outline here a preliminary classification of GLMS-measures in the basis of the papers surveyed in our state of the art analysis. What is really interesting at this point is that the measures can be classified according to several dimensions:

- Condition. We can measure elements that fulfil some condition regarding their attributes (*attribute conditional* measures, e.g. number of elements of a category), regarding some structural condition (*structural conditional* measures, e.g. number of nodes that have not edges stemming out) or regarding a property (*property conditional* measures, e.g. number of nodes that have a given value for a property). More than one condition may be checked in a given measure.
- Result. For a particular concept, we can compute the value as such (*absolute* measures), with several variations: counting, obtaining the maximum, distance, etc. We can divide this absolute value by a superconcept (*normalized* measures, e.g. number of elements of a category divided by the total number of elements) or we can compute a ratio compared to some other concept (*crossed* measures, e.g. number of nodes divided by number of edges). Also, sometimes we are more interested in getting the elements that apply for the computed concept that the result itself, allowing to use this measure as a filter for another (*filtering* measures, e.g. obtaining the set of elements that fulfil some structural condition).
- Input. The measures may be applied to a full diagram (*diagram* measures) or just to a part of it (*subdiagram* measures). This second case is often used after a filtering measure has restricted the diagram to some elements (probably of different categories). A particular case of the second type is when the measures apply to just one element (*individual* measures).

Table 3 presents a sample of the catalogue exploring different variations of a measure for counting elements that belong to a category. GLMS-measures are defined as operations specified in OCL [20]. M1 gets the set of elements of a particular category from a subdiagram. M2 is an M1's particularization in which the subdiagram is the full diagram. M3 is the one used in the previous section. M4 defines a property-conditional normalized measure over the diagram: since it depends on a property, the name of the property and the required value are added as parameters; since it is normalized, the special case of having no elements of the category has to be treated separately. M5 is an example of structural conditional measure that counts the number of nodes of a certain category *cat1* connected through an outgoing edge to nodes of another category *cat2*. M6 shows the combination of several condition types by counting the same than M5 but also checking property values in the involved elements. We may see how some measures can be defined on top of others, e.g. M3 on top of M2 and M2 on top of M1, making easier the definition process. In particular, M4, M5 and M6 show the combination of existing measures into one using filtering versions. We remark also the use of the following lexical pattern for naming measures according to their classification: condition-result-input, e.g. in M1, condition = byCategory, result = Filtering, input = Subdiagram.

Table 3. Some GLMS-measures. Unless otherwise stated, parameters are of type String. "SSN" stands for "Set(Sequence(Node))". The commented version of this OCL expressions can be found in Appendix 1.

M1. Attribute conditional, filtering, subdiagram

```
context Diagram::byCategoryFilteringSubdiagram(se:Set(Element),cat):Set(Element)
post: result = se->select(e | e.category->includes(cat))
```

M2. Attribute conditional, filtering, diagram

```
context Diagram::byCategoryFilteringDiagram(cat): Set(Element)
post: result = byCategoryFilteringSubdiagram(Element.allInstances(), cat)
```

M3. Attribute conditional, counting, diagram

```
context Diagram::byCategoryCountDiagram(cat): Integer
post: result = self.byCategoryFilteringDiagram(cat)->size()
```

M4. Property conditional, normalized, diagram

```
context Diagram::byCategoryByPropertyNormalizedDiagram(cat, np, val): Real
post: byCategoryCountDiagram(cat) = 0 implies result = 0
post: byCategoryCountDiagram(cat) > 0 implies
        result = byCategoryFilteringDiagram(cat)->
                    select(e | e.assignment->exists(a | a.property.name = np and
                                                        a.value = val))-> size()
                 / byCategoryCountDiagram(cat)
```

M5. Attribute and structural conditional, absolute, diagram

```
contextDiagram::byCategoryByTargetElementCategoryCountDiagram(cat1,cat2):Integer
post: result = byCategoryCountSubdiagram(
               byCategoryFilteringDiagram(cat1).targetEdge.targetElement, cat2)
```

M6. Attribute, structural and property conditional, absolute, individual

```
context Diagram::
         byCategoryAndPropertyByTargetElementCategoryAndPropertyCountDiagram
                           (cat1, prop1, val1, cat2, prop2, val2): Integer
post: result = byCategoryByPropertyCountSubdiagram(
                    byCategoryByPropertyFilteringDiagram(cat1, prop1, val1).
                        targetEdge.targetElement, cat2, prop2, val2)
```

M7. Attribute conditional, filtering, diagram

```
context Diagram::allPathsFilteringDiagram(catN, catE): SSN
post: result = Node.allInstances()->
           select(n | n.category->includes(catN) and
                      n.sourceEdge->select(e | e.category->includes(catE))
                                   ->isEmpty())
          ->iterate(x; s: SSN=Set{Sequence{}} | s->union(x.allPaths(catN, catE)))
context Node::allPaths(catN, catE): SSN
post: targetEdge->select(e | e.category->includes(catE))->isEmpty() and
        category->includes(catN) implies result = Set{Sequence{self}}
post: targetEdge->select(e | e.category->includes(catE))->notEmpty()
                 implies result =
 if category->includes(catN) then
  targetEdge->select(e | e.category->includes(catE)).targetElement->
  iterate(x; s: SSN=Set{Sequence{}} | s->union(inFront(x.allPaths(catN, catE),x))
 else targetEdge->select(e | e.category->includes(catE)).targetElement->
  iterate(x; s: SSN=Set{Sequence{}} | s->union(x.allPaths(catN, catE)) endif
```

M8. Attribute conditional, maximum, diagram

```
context Diagram::allPathsMaxDiagram(catN: String, catE: String): Integer
post: result = allPathsFilteringDiagram(catN, catE)->
             select(p | allPathsFilteringDiagram(catN, catE)->
                          forAll(p2 | p->size() >= p2->size())->size()
```

The last two measures illustrate a very different but also common type of measure for conceptual schemas. In M7 we define a filtering measure for generating all paths composed by nodes of a given category *catN* following edges of another category *catE*. It relies on an operation (*allPaths*) applied to the roots of the path. This operation is also shown, with two postconditions showing the case of final node (i.e., with no outgoing *catE* edges) and the recursion case, in which the current node is put in front to each (recursively-generated) path only if it is a *catN* node (*inFront* operation is not included for lack of space, it basically uses the *prepend* OCL operator to put the element in front of each sequence generated with an *iterate*). On top of M7, M8 computes the longest path comparing pair-wise all paths and keeping the longest as result.

5 Defining Measures over a Modeling Language Metaschema

In this section we illustrate how to define a measure suite for a particular conceptual modeling language which is based on the GLMS. We consider a concrete proposal of Business Process Modeling (BPM) notation, used by Balasubramanian and Gupta [4] in their formulation of a BPM measure suite. We already used this case study in a previous work [21] in the context of definition of measures in *i**, and from that experience we think it is a nice candidate to illustrate the framework presented here. As happened in that paper, for the sake of space, we focus on a representative subset of measures. Since the notation does not have name and we need one, we denote it by BPM-BG (after the authors' initials).

The method we propose is structured into two steps that are presented in the next two subsections:

- The conceptual modelling language metaschema has to be connected to the GLMS to allow proper inheritance of the GLMS-measures.
- The measure suite is defined as outlined in Section 3.

5.1 Refactoring the BPM-BG Metaschema

In the general case, the modelling language has an already defined metaschema. To apply our framework, we need first to adapt it to our needs. The refactoring of the language metaschema has the purpose of expressing its relevant concepts in terms of the GLMS classes. The refactoring is designed to keep the elements of the language's original metaschema and then adding new elements needed to adapt it to the GLMS. Moreover, the new elements will only include information that can be derived from those of the original metaschema.

BPM-BG proposes a 3-view model for business processes [4] but for the purposes of the paper we focus on one of them. The workflow view diagram reveals its set of constituent activities, their precedence relationships and the business participants (either human or system) that execute them. A workflow allows forks and merges. Activities have automation degrees depending on the degree of interaction system-user. Activities with human intervention (manual or interactive) may be discretional (i.e., humans make decisions in a non-fully controllable manner). Fig. 3 shows the BPM-BG workflow view metaschema. Integrity constraints exist but we do not show them for the sake of conciseness. A correspondence must be established to relate the concepts of the BPM-BG metaschema with GLMS classes (see Table 4).

Those concepts that are captured by classes of the BPM-BG metaschema (e.g. *WorkflowViewDiagram*, *Activity*) are directly defined as subclasses of their corresponding GLMS classes (e.g. *Activity* is declared as a subclass of *Node* and *Simple* and its attributes redefine the attributes *name* and *category* of *Element*).

On the other hand, there is a mismatch for those concepts captured by associations or attributes since they cannot be directly defined as subclasses. We need a more elaborated refactoring to allow the inducement of meta-measures into the BPM-BG metaschema. In the following, we describe the refactoring of the association *Precedes* and the attribute *automationDegree*.

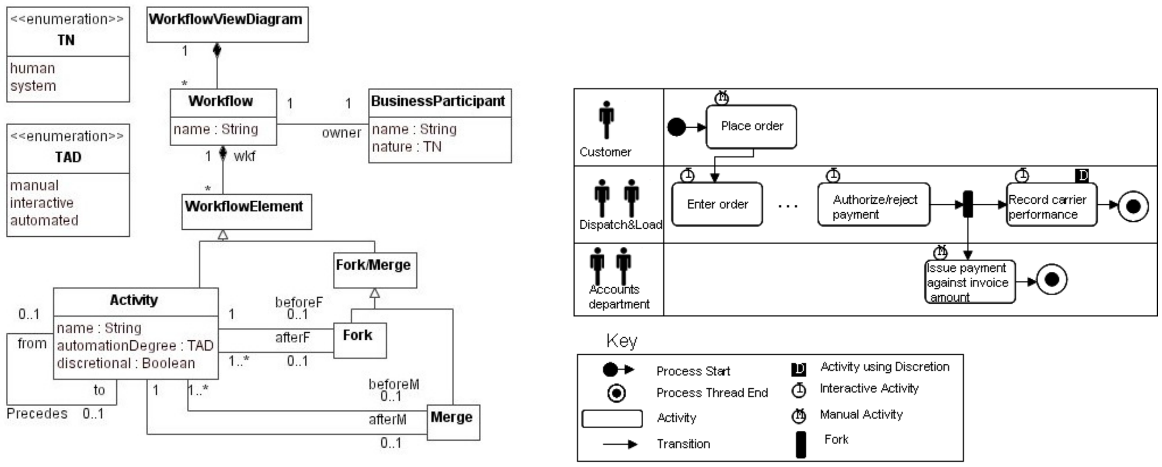
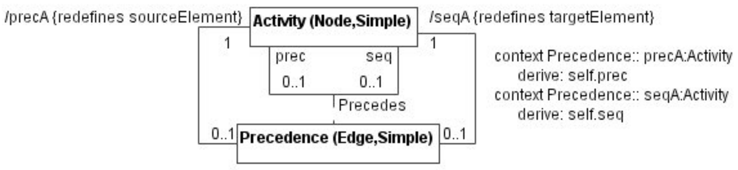


Fig. 3. Fragment of the BPM-BG workflow view metaschema and workflow view example

Table 4. Correspondence of BPM-BG metaschema and the GLMS.

BPM-BG metaschema concepts		GLMS classes
Concept	**Representation**	**Concept**
WorkflowViewDiagram	Class	Node, Diagram
Workflow	Class	Node, Compound
WorkflowElement, Activity, Fork/Merge, Fork, Merge, BusinessParticipant	Class	Node, Simple
beforeF, afterF, beforeM, afterM, Precedes	Association	Edge, Simple
automationDegree, discretional, nature	Attribute	Property

The refactoring of the association *Precedes* consists, basically, on specifying it as an association class (*Precedence*) which, at the same time, is defined as a subclass of *Edge* and *Simple* (its corresponding concepts in the GLMS). Fig. 4 depicts the refactoring of the *Precedes* association (where, for brevity, *(C)* beside the class name stands for its definition as subclass of *C*).

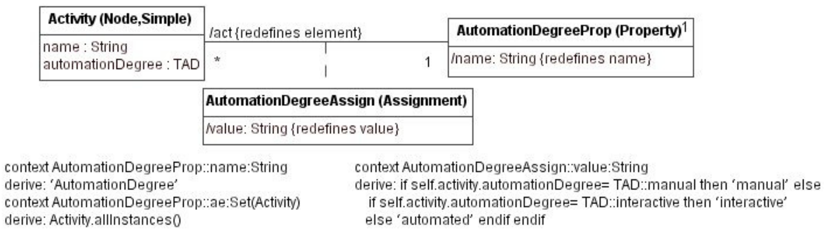


```
context Precedence:: precA:Activity
    derive: self.prec
context Precedence:: seqA:Activity
    derive: self.seq
```

Fig. 4. Refactoring of the *Precedes* association

Another aspect of the refactoring made in Fig. 4 that deserves attention is that two derived associations relating *Activity* and *Precedence* have been added such that they are calculated from the *Precedes* association. They also redefine the two associations between *Node* and *Edge* of the GLMS. As a consequence, the instances of *Precedes* are used to populate the redefined elements of the GLMS. In this way, the GLMS is populated with instances of the original elements of the BPM-BG metaschema.

Now, consider the attribute *automationDegree*. Its refactoring, illustrated in Fig. 5, consists, basically, on specifying a new singleton class *AutomationDegreeProp*, defined as a subclass of *Property* and a new association class *AutomationDegreeAssign*, defined as a subclass of *Assignment*, such that it relates *Activity* and *AutomationDegreeProp* through a derived association. This derived association redefines its corresponding GLMS association. The attributes of the new classes are also derived and redefine their corresponding GLMS attributes. By contrast, the *AutomationDegreeProp* singleton class itself is not derived since UML does not admit the general definition of derived classes [22] and, for this reason, we assume an initialization operation that creates its single instance.


Activity (Node,Simple)
name : String
automationDegree : TAD
/act {redefines element}
*
1
AutomationDegreeProp (Property)1
/name: String {redefines name}
AutomationDegreeAssign (Assignment)
/value: String {redefines value}

```
context AutomationDegreeProp::name:String
derive: 'AutomationDegree'
context AutomationDegreeProp::ae:Set(Activity)
derive: Activity.allInstances()
```

```
context AutomationDegreeAssign::value:String
derive: if self.activity.automationDegree= TAD::manual then 'manual' else
   if self.activity.automationDegree= TAD::interactive then 'interactive'
  else 'automated' endif endif
```

Fig. 5. Refactoring of the *automationDegree* attribute

The rest of non-class elements of the BPM-BG metaschema can be refactored in a similar way. As intended, the resulting BPM-BG refactored metaschema keeps all the elements from the original BPM-BG, although it is true that the Open-Closed Principle [23] is not fully applied due to the added subtyping relationships in the original classes. Thanks to the use of derived and redefined information, the whole metaschema (i.e., the combination of the GLMS and the BPM-BG metaschema) is populated from the instances of the original BPM-BG metaschema and thus the GLMS-measures are applicable over the refactored metaschema. To avoid the violation of the Open-Closed principle, for each class A of the language metaschema which inherits from two GLMS's *Element* subclasses B and C (e.g., *Activity* that inherits from *Node* and *Simple*), we could create a class A2 that inherits from A, B and C. Therefore the initial language metaschema would be really preserved. The main ideas behind our approach would not change in a significant way.

5.2 Inheriting Measures over the BPM-BG Metaschema

Once the two metaschemas have been aligned, we can define the measures over them. In general, we aim at simply invoking the operations inherited from the GLMS classes that define the GLMS-measures, but as it will be shown below, this is not always possible.

For the sake of brevity, we focus on 3 representative measures over the BPM workflow view. The first is a case of immediate application, the second requires a slight adaptation, whilst the third needs more work but still makes use of the metameasures.

- AAF. Proportion of total activities in a process that require system support. Indicator of throughput. The GLMS-measure M4 (see Table 3) to count the number of elements of a certain category (*Activity*) that fulfil a property (*automationDegree*) is applied twice and results added. This is an example of measure easy to obtain from the GLMS.

```
context WorkflowViewDiagram::AAF(): Real
post: result =
    byCategoryByPropertyNormalizedDiagram('Activity','automationDegree','automated')
  + byCategoryByPropertyNormalizedDiagram('Activity','automationDegree','interactive')
```

- APF. Longest path of activities that must be executed sequentially divided by the total number of activities. Indicator of throughput. It is based on the computation of paths formed by *Activities* using the *Precedence* relationship introduced when refactoring (Fig. 4) using the GLMS-measure M8. Its definition is straighforward:

```
context WorkflowViewDiagram::APF(): Real
post: byCategoryCountDiagram('Activity') = 0 implies result = 0
post: byCategoryCountDiagram('Activity') > 0 implies
      result = allPathsMaxDiagram('Activity', 'Precedence') / byCategoryCountDiagram('Activity')
```

- TDRF. Proportion of transitions of flow between business participants from system activities to human activities. Indicator of reliability. This is a complex measure that cannot be simply induced by GLMS-measures. We provide below a simplified version neither considering forks nor merges (which basically require repeating two additional times the given expression). The focus is on the different nature of activities using the GLMS-measure M6 in the filtering version, and the resulting elements need to be filtered again to discard those edges that do not represent transition of flows between business participants. We remark that for this second filtering, we work directly at the level of the BPM-BG metaschema, although we could have chosen to define a GLMS-measure if we had considered that the type of filter is interesting enough to be included at this level. Note that even in this case, the existence of GLMS-measures helps formalising the measure.

```
context WorkflowViewDiagram::TDRF(): Real
post: result = byCategoryAndPropertyByTargetElementCategoryAndPropertyFilteringDiagram
                   ('Activity','nature','system', 'Activity','nature','human')->
               select(e | e.wkf.owner <> e.to.wfk.owner)->size()
               / Activity.allInstances()->select(e | e.wkf.owner <> e.to.wfk.owner)->size()
```

5.3 Discussion: Relationship with MOF

Once the full proposal has been presented, a final reflection can be made about the modelling architecture that we have followed. Our framework proposes to circumscribe both the GLMS and the modelling language metaschema at the M2 level (according to the framework of a meta-modelling architecture defined in [19]. Another option could have been to keep the GLMS at the M2 level and to define the language metaschemas at the M1 layer as its instantiation. Our reasons for not following this latter approach are that: 1) it leaves no room for runtime instances since, then, M0 would represent specific schemas (for example, a BPM diagram, but not particular process instances), and 2) to define the language metaschema at the M1 layer may be counterintuitive and may damage the understandability of the approach. Other alternative could have been defining the GLMS at M3 level but then our approach would have not aligned with the four-level metamodel hierarchy that proposes MOF at the M3 level.

6 Validation

We have performed a two-fold validation of our approach. First, we have used a comprehensive catalogue of measures for a particular type of diagrams, UML class diagrams. We have used an extensive survey [12] that compiles 67 measures from several authors. The results are summarized in Table 5. The most interesting result is that 62% of the measures are direct applications of some GLMS-measure (most times just one, some times a bit more, similarly to AAF in Section 5.2), whilst other 26% require more complicated combinations but are still easy to do (e.g., CL1 that computes sets of responsibilities, that is a concept that has many refinements). 3% of the measures are derived, i.e. just a ratio of other more basic ones. The real hard ones represent the 6% of the population, which were so particular that it makes no sense to define a GLMS-measure abstracting their meaning. But even in this case, as happened with TDRF, all of them used some GLMS-measure as the basis for computation, for instance, NMO computes the total number of methods overridden by a subclass, which requires to generate paths of classes and for that purpose, M7 may be used. Finally, we remark that 6% of the measures are hard to define using the OCL since they involve square and square root computations. As additional information, 6% of all the measures required expert judgement (e.g., WMC requires an expert to weight the complexity of methods), which would be modelled as usual as properties in our framework.

Table 5. Adequacy of our framework for UML class diagrams measures.

How	How many	Which ones
Straightforward	41	WMC, DIT, NOC, DAC, DAC', NOM, PIM, NIM, NIV, NCM, NCV, NMI, NMA, APPM, PK3, OA1, OA2, OA3, OA5, DAM, DCC, MOA, DSC, NOH, ANA, NOP, NAssoc, NAgg, NDep, NGen, NGenH, NAggH, maxDIT, MaxHAgg, NAssocC, HAgg, NODP, NP, NW, MAgg, NDepIN, NDepOut
Require several combinations	17	MIF, AIF, PF, ACAIC, OCAIC, DCAEC, OCAEC, ACMIC, OCMIC, DCMEC, OCMEC, CL1, CL2, PK1, PK2, OA7, MFA
Derived	2	SIZE2, SIX
Specific	4	MHF, AHF, NMO, CAMC
Not-well suited	2	OA4, OA6

On the other hand, we have analysed some other types of diagrams with a sample of measures found in concrete proposals. In particular, we have explored the diagrams: ER, use case, activity, statechart, social network, *i**, in addition to the BPM case. To make the sample more representative, we have used different types of sources: scientific papers for the ER [24] and BPM [4] an existing tool for measure calculation, SDmetrics (http://www.sdmetrics.com/), for use cases, activity and statechart diagrams; and even the Wikipedia for social networks. The full description of these cases can be found in Appendix 2 and the results are summarised in Table 6. In total, we have analysed 45 measures that have required the application of 58 patterns of 17 different types (in the last row we show the types of different patterns considering the totality of metrics). The two next columns try to provide an indicator of the applicability of our approach. We may observe that up to 69% of the measures have been defined by simply invoking one metameasure and in addition 7% more have been defined by reusing some measures defined below (e.g., in social networks, degree centrality as the sum of in-degree and out-degree). We have defined an indicator to measure somehow the overall customization effort, shown in the last column: *x*+*y*+*z* means "*x* navigations, *y* operators on collections, *z* operators not on collections". Navigations include oclAsType (considered as "navigation" through a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators. It is worth to remark the case of social networks since it illustrates the fact that, in the discipline of software engineering, new models and notations continuously emerge for whatever reason and in particular social networks are becoming increasingly popular for different purposes, e.g. requirements prioritisation [25]. Our approach facilitates the definition of measures for these new approaches.

Table 6. Adequacy of our framework for different types of diagrams' measures.

Type of diagram	**Number of measures**	**Patterns applied**	**Types of patterns applied**	**Immediate measures**	**Measures by reuse**	**Customization effort**
ER	12	13	4	8 (67%)	1 (8%)	7+1+11
Activity	10	10	1	10 (100%)	0	0+0+0
Statechart	7	7	2	5 (71%)	1 (14%)	0+0+3
Use case	6	6	4	5 (83%)	0	4+0+1
Social network	5	9	6	2 (40%)	1 (20%)	0+2+8
BPM	5	13	8	1 (20%)	0	10+4+15
TOTAL	45	58	17	31 (69%)	3 (7%)	21+7+38

7 Conclusions and Future Work

In this paper we have argued about the possibility of defining measures for conceptual schema diagrams not in an ad-hoc way, but by manipulation of some generic measures that are adapted to the particular type of diagram after a metaschema alignment. We have presented methodological aspects of the proposal, a precise definition in terms of metaschema transformations, and a first validation step. The benefits of the proposal are: 1) simplification of measures definition: although measures over the GLMS metaschema may be complex, they are defined only once as a predefined catalogue and definition of specific measures on top of them is, in general, simple; 2) improvement of measure understandability; 3) ontological alignment since related measures can be defined on top of the same GLMS-measure; 4) possibility of defining the rationale of similar measures in a unified way at the GLMS-level; 5) facilitation of adapting measures over a modelling language to other languages. As drawbacks, we must point out the need of creating the initial catalogue and that, although not many (see Tables 5 and 6), some measures require still non-negligible work or even are too specific to be defined as GLMS-measure particularizations.

The future work is organized along three directions that need to be jointly run. First, validate further the approach by considering more types of models and more measures on them. Second, complete the catalogue of GLMS-measures. Third, develop tool support to facilitate both the process of managing the GLMS-measures catalogue and the process of browsing it when defining a new set of measures.

References

1. DeMarco T.: *Controlling Software Projects: Management, Measurement and Estimation*. Yourdon Press, 1986.
2. Jorgensen, M., Shepperd, M.: “A Systematic Review of Software Development Cost Estimation Studies”. *IEEE Transactions on Software Engineering*, 33(1), January 2007.
3. Si-Said Cherfi, S., Akoka, J., Comyn-Wattiau, I.: “Conceptual Modelling Quality - From EER to UML Schemas Evaluation”. ER 2002.
4. Balasubramanian, S., Gupta, M.: “Structural Metrics for Goal Based Business Process Design and Evaluation”. *Business Process Management Journal*, 11(6), 2005.
5. Chidamber, S.R., Kemerer, C.F.: “A Metrics Suite for Object Oriented Design”. *IEEE Transactions on Software Engineering*. 20(6), June 1994.
6. Múñoz, L., Mazón, J., Trujillo, J.: “A family of Experiments to validate Measures for UML Activity Diagrams of ETL Processes”. *Information and Software Technology*, 52(11), 2010.
7. Kim, H. et al. “Developing Software Metrics Applicable to UML Models”. QAOOSE 2002.
8. Vanderfeesten, I., Cardoso, J., Mendling, J., Reijers, H.A., van der Aalst, W.: “Quality Metrics for Business Process Models”. *BPM and Workflow Handbook,* 2007.
9. Sutcliffe, A., Minocha, S.: “Linking Business Modelling to Socio-technical System Design”. CAiSE 1999.
10. McQuillan, J.A., Jacqueline, Power, J.F.: “On the Application of Software Metrics to UML Models”. MoDELS 2006 Workshops.
11. Moody, D.: “Metrics for Evaluating the Quality of Entity Relationship Models”. ER 1998.
12. Genero, M., Piattini, M., Calero, C.: “A Survey of Metrics for UML Class Diagrams”.

Journal of Object Technology 4(9), 2005.
13. Genero, M., Miranda, D., Piattini, M.: “Defining and Validating Metrics for UML Statechart Diagrams”. QAOOSE 2002.
14. Grau, G., Franch, X., Maiden, N. "PR*i*M: an *i**-based Process Reengineering Method for Information Systems Specification". *Information and Software Technology* 50, (1-2), 2008.
15. Franch, X.: “On the Quantitative Analysis of Agent-Oriented Models”. CAiSE 2006.
16. Habra, N., Abran, A., Lopez, M., Sellami, A.: “A Framework for the Design and Verification of Software Measurement Methods”. *Journal of Systems and Software*, 81(5), 2008.
17. Bondy A., Murty U. S. R.: *Graph Theory*. Springer, 2008.
18. Rolland, C., Salinesi, C., Etien, A.: “Eliciting Gaps in Requirements Change”. *Requirements Engineering Journal* 9(1), 2004.
19. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.3. OMG Document Number: formal/2010-05-03, 2005.
20. Object Management Group. Object Constraint Language (OCL), Version 2.2. Available Specification (formal/2010-02-01), 2010.
21. Franch, X.: “A Method for the Definition of Metrics over *i** Models”. CAiSE 2009.
22. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.3, (formal/2010-05-05), 2010.
23. Martin, R.C. *Agile Software Development*. Prentice-Hall, 2003.
24. Genero, M., Poels, G., Piattini, M.: “Defining and validating Metrics for assessing the Understandability of ER Diagrams”. *Data & Knowledge Engineering*, 64(3), 2008.
25. Lim, S.L., Quercia, D., Finkelstein, A. “StakeNet: using Social Networks to analyse the Stakeholders of Large-scale Software Projects”. ICSE 2010.

Appendix 1

Table 3 presents a sample of GLMS-measures. They are defined as operations specified in OCL over the Graph-like metaschema (GLMS) shown in Figure 1.

In the following, we comment table 3 measures. Unless otherwise stated, parameters of the measures are of type String and "SSN" stands for "Set(Sequence(Node))".

Measure M1 gets the set of elements of a particular category from a subdiagram:

```
   context
Diagram::byCategoryFilteringSubdiagram(se:Set(Element),cat):Set(Element)
   post: result = se->select(e | e.category->includes(cat))
```

Parameter *se* corresponds to the set of elements of the departing subdiagram and parameter *cat* gives the category of the elements to get. The OCL expression uses the operator *select* to obtain the elements of the subdiagram such that their attribute *category* includes the value *cat*.

Measure M2 is an M1's particularization in which the subdiagram is the full diagram.

```
   context Diagram::byCategoryFilteringDiagram(cat): Set(Element)
   post:  result  =  byCategoryFilteringSubdiagram(Element.allInstances(),
cat)
```

Now, the only parameter is *cat* to give the category of the elements to get from the full diagram. The OCL expression uses M1 with the set of all the elements of the diagram as first parameter (the OCL operator *allInstances* is used to obtain all the instances of the class *Element*).

M3 counts the number of occurrences of a particular category *cat* of element in a diagram:

```
   context Diagram::byCategoryCountDiagram(cat): Integer
   post: result = self.byCategoryFilteringDiagram(cat)->size()
```

This OCL expression uses M2 to obtain all the elements of category *cat* of the diagram and, then, it uses the operator *size* to obtain their total number.

M4 defines a property-conditional normalized measure over the diagram:

```
   context Diagram::byCategoryByPropertyNormalizedDiagram(cat, np, val):
Real
   post: byCategoryCountDiagram(cat) = 0 implies result = 0
   post: byCategoryCountDiagram(cat) > 0 implies
         result = byCategoryFilteringDiagram(cat)->
            select(e | e.assignment->exists(a | a.property.name = np and
                         a.value = val))-> size()
                       / byCategoryCountDiagram(cat)
```

Since it depends on a property, the name of the property (*np*) and the required value (*val*) are added as parameters. Since it is normalized, the special case of having no elements of the category has to be treated separately in the first postcondition (*post*) of the operation. The second postcondition deals with the non-empty case. It uses M2 to obtain all the elements of category *cat* of the diagram; and then, it uses the operator *select* to filter those elements with value *val* for property *np* and the operator *size* to count them. The obtained number of elements is divided by the total number of elements of the category *cat* in the diagram which is obtained using M3.

M5 counts the number of nodes of a certain category *cat1* connected through an outgoing edge to nodes of another category *cat2*.

```
   contextDiagram::byCategoryByTargetElementCategoryCountDiagram(cat1,ca
t2):Integer
   post: result =
   byCategoryCountSubdiagram(byCategoryFilteringDiagram(cat1).targetEdge
.targetElement, cat2)
```

The OCL expression uses an auxiliary measure *byCategoryCountSubdiagram* which counts the number of elements of a category (*cat2*) of a subdiagram. The subdiagram must have the set of elements that receive an outgoing edge from elements with category *cat1*. This is obtained by using measure M2 that gives the elements of category *cat1* of the full diagram followed by two navigations (*targetEdge.targetElement*) that obtain the target nodes connected to the outgoing edges of those elements.

M6 counts the same as M5 but also checking property values in the involved elements.

```
context Diagram::
byCategoryAndPropertyByTargetElementCategoryAndPropertyCountDiagram
                             (cat1, prop1, val1, cat2, prop2, val2): Integer
post: result = byCategoryByPropertyCountSubdiagram(
            byCategoryByPropertyFilteringDiagram(cat1, prop1, val1).
                       targetEdge.targetElement, cat2, prop2, val2)
```

This postcondition uses two auxiliary measures. First, measure *byCategoryByPropertyCountSubdiagram* which counts the number of elements of a category (*cat2*) of a subdiagram that have a given value (*val2*) for a given property (*prop2*). Additionally, to obtain that subdiagram, another auxiliary measure *byCategoryByPropertyFilteringDiagram* is used. It gives the set of elements of a category (*cat1*) of a diagram that have a given value (*val1*) for a given property (*prop1*).

M7 is filtering measure for generating all paths composed by nodes of a given category *catN* following edges of another category *catE*.

```
context Diagram::allPathsFilteringDiagram(catN, catE): SSN
post: result = Node.allInstances()->
         select(n | n.category->includes(catN) and
                    n.sourceEdge->select(e | e.category->includes(catE))
```

```
                                  ->isEmpty())
    ->iterate(x; s: SSN=Set{Sequence{}} | s->union(x.allPaths(catN, catE)))
```

The operation generates first the nodes *catN* that are starting point of such paths, by checking that there are no edge *catE* pointing to such node *catN*. For all of the nodes that fulfill this condition, all the paths that start from that node are generated using an auxiliary operation *allPaths*. The generated paths are put together in the result. The *allPaths* operation is also shown below:

```
      context Node::allPaths(catN, catE): SSN
      post: targetEdge->select(e | e.category->includes(catE))->isEmpty()
        and
         category->includes(catN) implies result = Set{Sequence{self}}
      post: targetEdge->select(e | e.category->includes(catE))->notEmpty()
                 implies result =
   if category->includes(catN) then
     targetEdge->select(e | e.category->includes(catE)).targetElement->
     iterate(x; s: SSN=Set{Sequence{}} | s->union(inFront(x.allPaths(catN,
        catE),x))
   else targetEdge->select(e | e.category->includes(catE)).targetElement->
     iterate(x; s:SSN=Set{Sequence{}} | s->union(x.allPaths(catN, catE))
  endif
```

It has two postconditions showing first the case of final node (i.e., with no outgoing *catE* edges) and the recursion case, in which the current node is put in front to each (recursively-generated) path only if it is a *catN* node (*inFront* operation is not included for lack of space, it basically uses the *prepend* OCL operator to put the element in front of each sequence generated with an *iterate*).

M8 computes the longest path comparing pair-wise all paths and keeping the longest as result. It is quite simple using the measure M7 above: it generates all the paths with the *catN* and *catE* as above, and selects the one that has the greatest size (i.e., the longest path), keeping then the size as result.

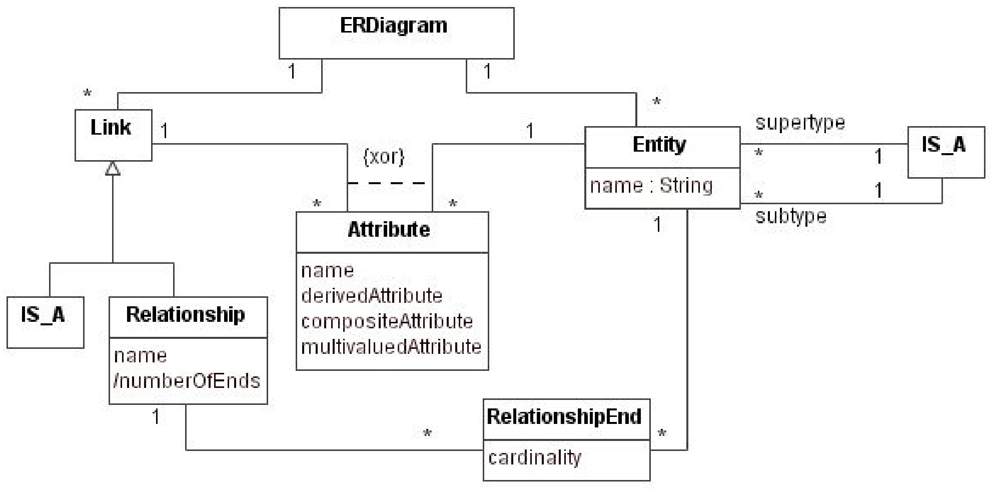
```
  context Diagram::allPathsMaxDiagram(catN: String, catE: String): Integer
  post: result = allPathsFilteringDiagram(catN, catE)->
            select(p | allPathsFilteringDiagram(catN, catE)->
                    forAll(p2 | p->size() >= p2->size())->size()
```

Appendix 2: Validation

ER measures

SOURCE: Marcela Genero Geert Poels, Mario Piattini: "Defining and validating metrics for assessing the understandability of entity–relationship diagrams". *Data & Knowledge Engineering*, 64(3), March 2008, pages 534-557.

FRAGMENT OF THE ER METASCHEMA



```
context Relationship::numberOfEnds:Integer
derive: self.relationshipEnd->size()
```

METAMODEL CORRESPONDENCE:

ERDiagram	Diagram
Entity	Node, Simple
Attribute	Node, Simple
Relationship	Edge, Simple
RelationshipEnd	Edge, Simple (from relationship to entity)
DerivedAttribute	Property (of attribute)
CompositeAttribute	Property (of attribute)
MultivaluedAttribute	Property (of attribute)

Cardinality	Property (of relationship end)
NumberOfEnds	Property (of relationship)
IS_A	Edge, Simple

CATALOGUE OF MEASURES

Measure: NE

Definition: The Number of Entities metric is defined as the number of entities within an ER diagram, considering both weak and strong entities

Formalization:

```
context ERDiagram::NE(): Integer
post: byCategoryCountDiagram('Entity')
```

Measure: NA

Definition: The Number of Attributes metric is defined as the total number of attributes defined within an ER diagram, taking into account not only entity attributes but also relationship attributes. In this number all attributes are included (but not the composing parts of composite attributes).

Formalization:

```
context ERDiagram::NA(): Integer
post: byCategoryCountDiagram('Attribute')
```

Measure: NDA

Definition: The Number of Derived Attributes metric is defined as the number of derived attributes within an ER diagram. The value of NDA is always strictly less than the value of NA.

Formalization:

```
context ERDiagram::NDA(): Integer
post:
byCategoryByPropertyCountDiagram('Attribute','derivedAttr
ibute','derived')
```

Measure: NCA

Definition: The Number of Composite Attributes metric is defined as the number of composite attributes within an ER diagram. This value is less than or equal to the NA value.

Formalization:

```
  context ERDiagram::NCA(): Integer
  post:
byCategoryByPropertyCountDiagram('Attribute','compositeAt
tribute','composite')
```

Measure: NMVA

Definition: The Number of Multivalued Attributes metric is defined as the number of multivalued attributes within an ER diagram. Again, this value is less than or equal to the NA value.

Formalization:

```
  context ERDiagram::NMVA(): Integer
  post:
byCategoryByPropertyCountDiagram('Attribute','multivalued
Attribute','multivalued')
```

Measure: NR

Definition: The Number of Relationships metric is defined as the total number of relationships within an ER diagram, excluding ISA relationships.

Formalization:

```
  context ERDiagram::NR(): Integer
  post: byCategoryCountDiagram('Relationship')
```

Measure: NM:NR

Definition: The Number of M:N Relationships metric is defined as the number of M:N relationships within an ER diagram. The value of NM:NR is less than or equal to the NR value.

Formalization:

```
  context ERDiagram::NMNR(): Integer
  post: byCategoryCountDiagram('Relationship') -
  byCategoryByPropertyFilteringDiagram('RelationshipEnd',
  'cardinality','1').sourceElement->asSet()->size()
```

Explanation: The OCL expression obtains the number of M:N Relationships by subtracting to the total number of relationships, the number of relationships that have at least one relationship end with a cardinality of 1. When calculating this last number, the operation *asSet* is used to avoid counting more than once the relationships that have several ends with cardinality 1.

Measure: N1:NR

Definition: The Number of 1:N Relationships metric is defined as the total number of 1:N and 1:1 relationships within an ER diagram. Also this value is less than or equal to the NR value. The number of 1:1 relationships is not used as a separate metric because these relationships are considered a subset of the 1:N relationships

Formalization:

```
  context ERDiagram::N1NR(): Integer
  post:
byCategoryByPropertyFilteringDiagram('RelationshipEnd',
  'cardinality','1').sourceElement->asSet()->size()
```

Explanation: The OCL expression obtains the number of relationships that have at least one relationship end with a cardinality of 1. The operation *asSet* is used to avoid counting more than once the relationships that have several ends with cardinality 1.

Measure: NBinaryR

Definition: The Number of Binary Relationships metric is defined as the number of binary relationships within an ER diagram. Again, the value is less than or equal to the NR value.

Formalization:

```
context ERDiagram::NBinaryR(): Integer
post: byCategoryByPropertyCountDiagram('Relationship',
'numberOfEnds','2')
```

Explanation: The derived attribute numberOfEnds has been to the ER metaschema to facilitate the definition of this measure.

Measure: NN_AryR

Definition: The Number of N-Ary Relationships metric is defined as the number of N-Ary relationships within an ER diagram. Its value is less than or equal to the NR value.

Formalization:

```
context ERDiagram::NN_AryR(): Integer
post: byCategoryCountDiagram('Relationship')-
      NBinaryR()
```

Measure: NRefR

Definition: The Number of Reflexive Relationships metric is defined as the number of reflexive relationships within an ER diagram. Its value is less than or equal to the value of the NR metric.

Formalization:

```
context ERDiagram::NRefR(): Integer
post: byCategoryFilteringDiagram('Relationship')->
     select(r|r.oclAsType(Relationship).relationshipEnd.
entity->asSet()->size() <
r.oclAsType(Relationship).relationshipEnd->size())->
size()
```

Explanation: The OCL expression calculates the number of reflexive relationships by selecting those relationships that have a number of ends with different entities (*asSet* eliminates the duplicates) less than its total number of ends.

Measure: NIS_AR

Definition: NIS_AR The Number of IS_A Relationships metric is defined as the number of IS_A relationships within an ER diagram. In this case, we consider one relationship for each super-type/sub-type pair.

Formalization:

```
context ERDiagram::NIS_AR(): Integer
post: byCategoryCountDiagram('IS_A')
```

SUMMARY:

Patterns applied, individual:

byCategoryCountDiagram	6
byCategoryByPropertyCountDiagram	4
byCategoryByPropertyFilteringDiagram	2
byCategoryFilteringDiagram	1

Patterns applied, by category:

Condition	
Attribute conditional	7
Structural conditional	
Property conditional	6
Result	
Absolute	10
Normalized	
Crossed	

Filtering	3
Input	
Full diagram	13
Subdiagram	
Individual	

Measures complexity:

Measure	Patterns applied	Different patterns applied	Expressions not covered by patterns (*)	Measures reused
NE	1	1	0	0
NA	1	1	0	0
NDA	1	1	0	0
NCA	1	1	0	0
NMVA	1	1	0	0
NR	1	1	0	0
NM:NR	2	2	1+0+3	0
N1:NR	1	1	1+0+2	0
NBinaryR	1	1	0	0
NN_AryR	1	1	0+0+1	1
NRefR	1	1	5+1+5	0
NIS_AR	1	1	0	0

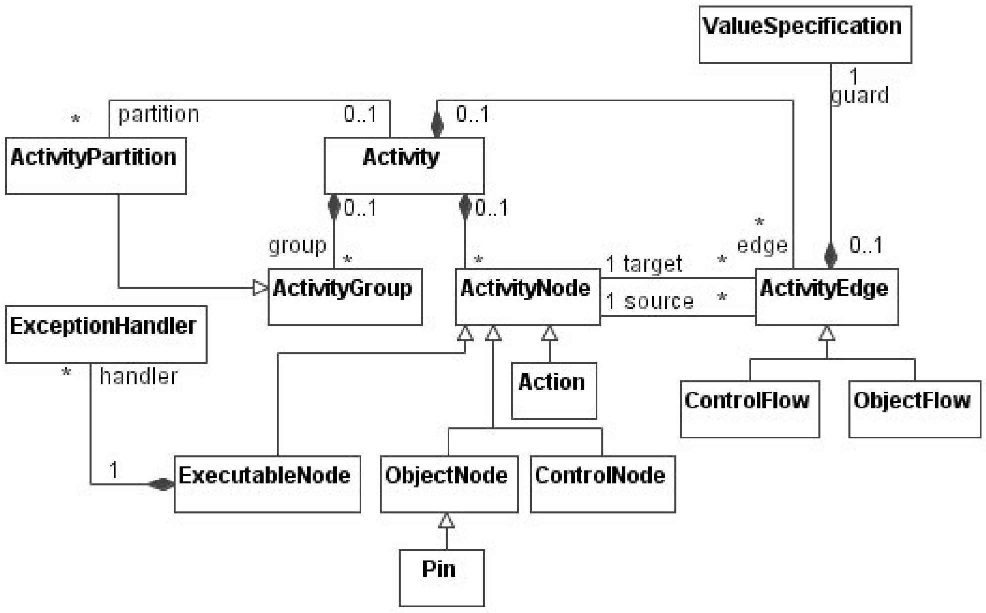
(*) x+y+z means "x navigations, y operators on collections, z operators not on collections". Navigations include oclAsType (considered as "navigation" in a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators.

Activity measures

SOURCE: http://www.sdmetrics.com/

FRAGMENT OF THE UML METASCHEMA

This fragment of the UML metaschema has been obtained from:
Object Management Group, OMG Unified Modeling Language (OMG UML), Superstructure, V2.3, (formal/2010-05-05), http://www.omg.org/spec/UML/2.3/ Superstructure/PDF/2010



METAMODEL CORRESPONDENCE:

Activity	Diagram
ActivityNode	Node, Simple
Action	Node, Simple
ObjecteNode	Node, Simple
ControlNode	Node, Simple
Pin	Node, Simple
ActivityPartition	Node, Simple
Partition	Edge, Simple (from Activity to ActivityPartition)
ActivityGroup	Node, Simple

Group	Edge, Simple (from Activity to ActivityGroup)
ActivityEdge	Edge, Simple (from ActivityNode to ActivityNode)
ControlFlow	Edge, Simple
ObjectFlow	Edge, Simple
Edge	Edge, Simple (from Activity to ActivityEdge)
ValueSpecification	Node, Simple
Guard	Edge, Simple (from ActivityEdge to ValueSpecification)
ExecutableNode	Node, Simple
ExceptionHandler	Node, Simple
Handler	Edge, Simple (from ExecutableNode to ExceptionHandler)

CATALOGUE OF MEASURES

Measure: Actions
Definition: The number of actions of the activity.

Formalization:

```
context Activity::Actions(): Integer
post: byCategoryCountDiagram('Action')
```

Measure: ObjectNodes
Definition: The number of object nodes of the activity.

Formalization:

```
context Activity::ObjectNodes(): Integer
post:     byCategoryCountDiagram('ObjectNode')
```

Measure: Pins
Definition: The number of pins on nodes of the activity.

Formalization:

```
context Activity::Pins(): Integer
post:     byCategoryCountDiagram('Pin')
```

Measure: ControlNodes
Definition: The number of control nodes of the activity.

Formalization:

```
context Activity::ControlNodes(): Integer
post:     byCategoryCountDiagram('ControlNodes')
```

Measure: Partitions
Definition: The number of activity partitions of the activity.

Formalization:

```
context Activity::Partitions(): Integer
post:     byCategoryCountDiagram('Partition')
```

Measure: Groups
Definition: The number of activity groups or regions of the activity.

Formalization:

```
context Activity::Groups(): Integer
post:     byCategoryCountDiagram('Group')
```

Measure: ControlFlows
Definition: The number of control flows of the activity.

Formalization:

```
context Activity::ControlFlows(): Integer
post:     byCategoryCountDiagram('ControlFlow')
```

Measure: ObjectFlows
Definition: The number of object flows of the activity.

Formalization:

```
context Activity::ObjectFlows(): Integer
post:     byCategoryCountDiagram('ObjectFlow')
```

Measure: Guards
Definition: The number guards defined on object and control flows of the activity.

Formalization:

```
context Activity::Guards(): Integer
post:     byCategoryCountDiagram('Guard')
```

Measure: ExcHandlers
Definition: The number of exception handlers of the activity.

Formalization:

```
context Activity::ExcHandlers(): Integer
post:     byCategoryCountDiagram('Handler')
```

SUMMARY:

Patterns applied, invididual:

byCategoryCountDiagram	10

Patterns applied, by category:

Condition	
Attribute conditional	10
Structural conditional	
Property conditional	
Result	
Absolute	10
Normalized	
Crossed	
Filtering	
Input	
Full diagram	10
Subdiagram	
Individual	

Measures complexity:

Measure	Patterns applied	Different patterns applied	Expressions not covered by patterns (*)	Measures reused
Actions	1	1	0	0
ObjectNodes	1	1	0	0
Pins	1	1	0	0
ControlNodes	1	1	0	0
Partitions	1	1	0	0
Groups	1	1	0	0
ControlFlows	1	1	0	0

ObjectFlows	1	1	0	0
Guards	1	1	0	0
ExcHandlers	1	1	0	0

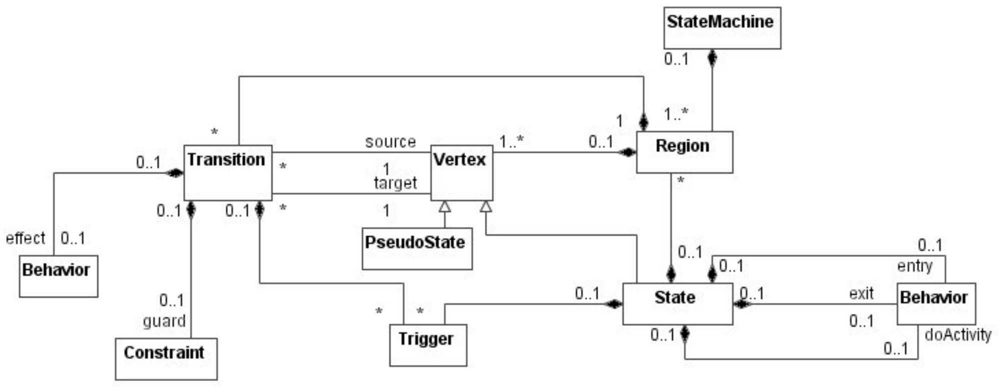
(*) x+y+z means “x navigations, y operators on collections, z operators not on collections”. Navigations include oclAsType (considered as “navigation” in a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators.

Statechart measures

SOURCE: http://www.sdmetrics.com/

FRAGMENT OF THE UML METASCHEMA

This fragment of the UML metaschema has been obtained from:
Object Management Group, OMG Unified Modeling Language (OMG UML), Superstructure, V2.3, (formal/2010-05-05), http://www.omg.org/spec/UML/2.3/ Superstructure/PDF/2010



METAMODEL CORRESPONDENCE:

StateMachine	Diagram
Vertex	Node, Simple
State	Node, Compound
Pseudostate	Node, Simple
Transition	Edge, Simple (from vertex to vertex)
Behavior	Node, Simple
Effect	Edge, Simple (from transition to behaviour)
Constraint	Node, Simple
Guard	Edge, Simple (from transition to constraint)
Trigger	Node, Simple
Entry	Edge, Simple (from state to behaviour)
Exit	Edge, Simple (from state to behaviour)
DoActivity	Edge, Simple (from state to behaviour)

Note: The metamodel correspondence does not include elements of the metaschema which are not relevant for the catalogue of measures.

CATALOGUE OF MEASURES

Measure: Trans
Definition: The number of transitions in the state machine.

Formalization:

```
context SMDiagram::Trans(): Integer
post: byCategoryCountDiagram('Transition')
```

Measure: TEffects
Definition: The number of transitions with an effect in the state machine.

Formalization:

```
context SMDiagram::TEffects(): Integer
post:
    byCategoryByTargetElementCategoryCountDiagram('Tran
sition','Behavior')
```

Measure: TGuard
Definition: The number of transitions with a guard in the state machine.

Formalization:

```
context SMDiagram::TGuard(): Integer
post:
byCategoryByTargetElementCategoryCountDiagram('Transition
','Constraint')
```

Measure: TTrigger
Definition: The number of triggers of the transitions of the state machine.

Formalization:

```
context SMDiagram::Ttrigger(): Integer
post:  byCategoryCountDiagram('Trigger')
```

Measure: States
Definition: The number of states in the state machine. This includes pseudo states, as well as composite and concurrent states of the statemachine, and recursively the states they contain, at all levels of nesting.

Formalization:

```
context SMDiagram::States(): Integer
post: byCategoryCountDiagram('State')+
byCategoryCountDiagram('Pseudostate')
```

Measure: SActivity
Definition: The number of activities defined for the states of the state machine. This counts entry, exit, and do activities defined for the states.

Formalization:

```
context SMDiagram::SActivity(): Integer
post:
byCategoryByTargetElementCategoryCountDiagram('State',
'Behavior')
```

Measure: CC
Definition: The cyclomatic complexity of the state-machine graph. This is calculated as Trans-States+2.

Formalization:

```
context SMDiagram::CC(): Integer
post: Trans() - States() + 2
```

SUMMARY:

Patterns applied, invididual:

byCategoryCountDiagram	4
byCategoryByTargetElementCategoryCountDiagram	3

Patterns applied, by category:

Condition	
Attribute conditional	4
Structural conditional	3
Property conditional	
Result	
Absolute	7
Normalized	
Crossed	
Filtering	
Input	
Full diagram	7
Subdiagram	
Individual	

Measures complexity:

Measure	Patterns applied	Different patterns applied	Expressions not covered by patterns (*)	Measures reused
Trans	1	1	0	0
TEffects	1	1	0	0
TGuard	1	1	0	0
TTrigger	1	1	0	0
States	2	1	0+0+1	0
SActivity	1	1	0	0
CC	0	0	0+0+2	2

(*) x+y+z means "x navigations, y operators on collections, z operators not on collections". Navigations include oclAsType (considered as "navigation" in a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators.

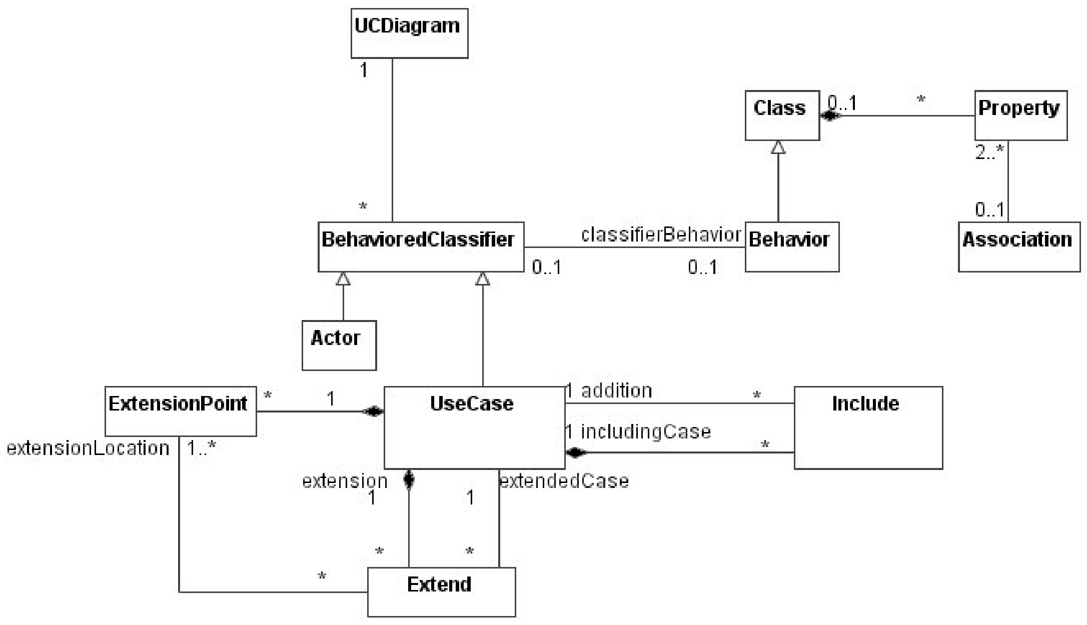
Use case measures

SOURCE: http://www.sdmetrics.com/

FRAGMENT OF THE UML METASCHEMA

This fragment of the UML metaschema has been obtained from:
Object Management Group, OMG Unified Modeling Language (OMG UML), Superstructure, V2.3, (formal/2010-05-05), http://www.omg.org/spec/UML/2.3/Superstructure/PDF/2010

The class *UCDiagram* has been added to it for compatibility with the GLMS



Constraint: Use cases can only be involved in binary Associations.

METAMODEL CORRESPONDENCE:

UCDiagram	Diagram
UseCase	Node, Simple
Actor	Node, Simple
Include	Edge, Simple (from includingCase use case to addition use case)
Extend	Edge, Simple (from extension use case to extendedCase use case)

ExtensionPoint	Node, Simple
ExtensionPoint-UseCase	Edge, Simple (from use case to extension point)
Association	Edge, Simple (from property to property, recall use cases can only be involved in binary associations)
BehavioredClassifier	Node, Simple
Behavior	Node, Simple
Class	Node, Simple
Property	Node, Simple

CATALOGUE OF MEASURES

Measure: NumAss
Definition: The number of associations the use case (with name *uc*) participates in.

Formalization:

```
  context UCDiagram::NumAss(uc: String): Integer
  post:
byCategoryByPropertyFilteringDiagram('UseCase','name',uc)
.oclAsType(UseCase).classifierBehavior.property.
  association->size()
```

Measure: ExtPts
Definition: The number of extension points of the use case.

Formalization:

```
  context UCDiagram::ExtPts(uc: String): Integer
  post:
byCategoryByTargetElementCategoryCountIndividual('UseCase
','ExtensionPoint',uc)
```

Measure: Including
Definition: The number of use cases which this one includes.

Formalization:

```
  context UCDiagram::Including(uc:String): Integer
  post:
byCategoryByOutgoingEdgeCategoryCountIndividual('UseCase'
,'Include',uc)
```

Measure: Included
Definition: The number of use cases which include this one.

Formalization:

```
context UCDiagram::Included(uc:String): Integer
post:
byCategoryByIncomingEdgeCategoryCountIndividual('UseCase'
,'Include',uc)
```

Measure: Extended
Definition: The number of use cases which extend this one.

Formalization:

```
context UCDiagram::Extended(uc:String): Integer
post:
byCategoryByIncomingEdgeCategoryCountIndividual('UseCase'
,'Extend',uc)
```

Measure: Extending
Definition: The number of use cases which this one extends.

Formalization:

```
context UCDiagram::Extending(uc:String): Integer
post:
byCategoryByOutgoingEdgeCategoryCountIndividual('UseCase'
,'Extend',uc)
```

Measure: Diags
Definition: The number of times the use case appears on a diagram.

Formalization: This measure cannot be represented in the UML metamodel.

SUMMARY:

Patterns applied, individual:

byCategoryByPropertyFilteringDiagram	1
byCategoryByTargetElementCategoryCountIndividual	1
byCategoryByOutgoingEdgeCategoryCountIndividual	2
byCategoryByIncomingEdgeCategoryCountIndividual	2

Patterns applied, by category:

Condition	
Attribute conditional	1
Structural conditional	5
Property conditional	
Result	
Absolute	5
Normalized	
Crossed	
Filtering	1
Input	
Full diagram	1
Subdiagram	
Individual	5

Measures complexity:

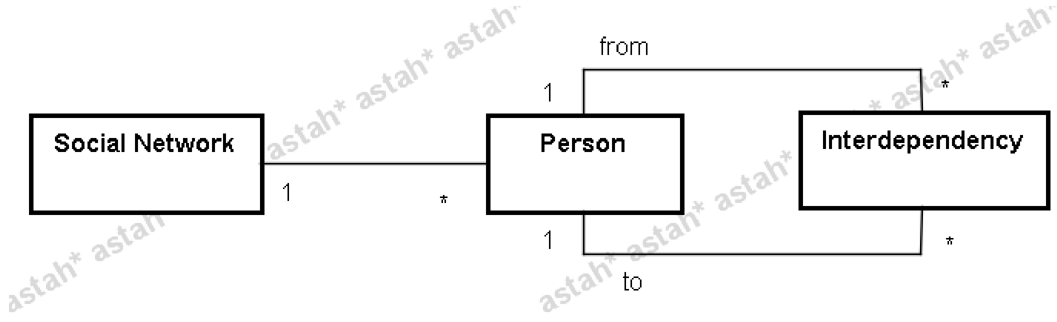
Measure	Patterns applied	Different patterns applied	Expressions not covered by patterns (*)	Measures reused
NumAss	1	1	4+0+1	0
ExtPts	1	1	0	0
Including	1	1	0	0
Included	1	1	0	0
Extended	1	1	0	0
Extending	1	1	0	0
Diags	---	------	-----	----

(*) x+y+z means "x navigations, y operators on collections, z operators not on collections". Navigations include oclAsType (considered as "navigation" in a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators.

Social network measures

SOURCE: Wikipedia, http://en.wikipedia.org/wiki/Social_network.

SOCIAL NETWORK METASCHEMA



METAMODEL CORRESPONDENCE:

SocialNetwork	Diagram, Compound
Person	Node, Simple
Interdependency	Edge, Simple

CATALOGUE OF MEASURES

Measure: Betweenness Centrality

Definition: The extent to which an individual lies between other individuals in the network. This measure takes into account the connectivity of the individual's neighbors, giving a higher value for individuals which bridge clusters. The measure reflects the number of individuals who an individual is connecting indirectly through their direct interdependencies.

(Note: we change “individual” to “person” to avoid confusion with the term “individual” used in our framework.)

Formalization:

```
  context        SocialNetwork::betweennessCentrality(stk:
String): Integer
  post: byCategoryFilteringIndividual(“Person”)->
        iterate(p1; ratio = 0 |
                byCategoryFilteringInvidual(“Person”)->
                forAll(p2 |
                       r = r +
                           if allPathsFilteringPairs("Person",
  "Interdependency", p1, p2) > 0
                         then
  allPathsFilteringPairs("Person", "Interdependency", p1, p2)
  ->filter(stk)->size() /
    allPathsCountPairs("Person", "Interdependency", p1, p2)))
                            else 0 endif
```

Explanation: The definition is a bit complicated due to the inability of the "iterate" operator to deal with pair of elements. Thus, it becomes necessary to add an aditional "forAll" inside each iteration to define the second element of reference to compute shortest paths. For each pair of elements, it is necessary to check the case that there are not paths among them (division by zero avoided). The "filter" operation is part of the vocabulary on paths of our pattern catalogue.

Measure: InDegree Centrality

Definition: Counts the number of incoming direct connections that a Person with name *stk* has

Formalization:

```
  context SocialNetwork::indegreeCentrality(stk: String):
Integer
  post:
byCategoryByIncomingEdgeCategoryCountIndividual("Person",
"Interdependency", stk)
```

Measure: OutDegree Centrality

Definition: Counts the number of outgoing direct connections that a Person with name *stk* has

Formalization:

```
  Context    SocialNetwork::outdegreeCentrality    (stk:
String): Integer
  post:
byCategoryByOutgoingEdgeCategoryCountIndividual("Person",
"Interdependency", stk)
```

Measure: Degree Centrality

Definition: Counts the number of direct connections that a Person with name *stk* has (both incoming and outgoing)

Formalization:

```
  context   SocialNetwork::degreeCentrality(stk:   String):
Integer
  post: indegreeCentrality(stk) + outdegreeCentrality(stk)
```

Measure: Closeness Centrality

Definition: Computes the inverse of the average of the distance from one Person with name stk to all other reachable Individuals in the network

Formalization:

```
  context          SocialNetwork::closenessCentrality(stk:
String): Integer
  post:  byCategoryCountModel("Person")-1 /
         allPathsMaxIndividual("Person",
"Interdependency", stk)
```

SUMMARY:

Patterns applied, invididual:

byCategoryFilteringIndividual	2
allPairsFilteringPairs	2
allPairsCountPairs	1
byCategoryByIncomingEdgeCategoryCountIndividual	2
byCategoryCountModel	1
allPathsMaxIndividual	1

Patterns applied, by category:

Condition	
Attribute conditional	7
Structural conditional	2
Property conditional	
Result	
Absolute	5
Normalized	
Crossed	
Filtering	4
Input	
Full diagram	1
Subdiagram	3
Individual	5

Measures complexity:

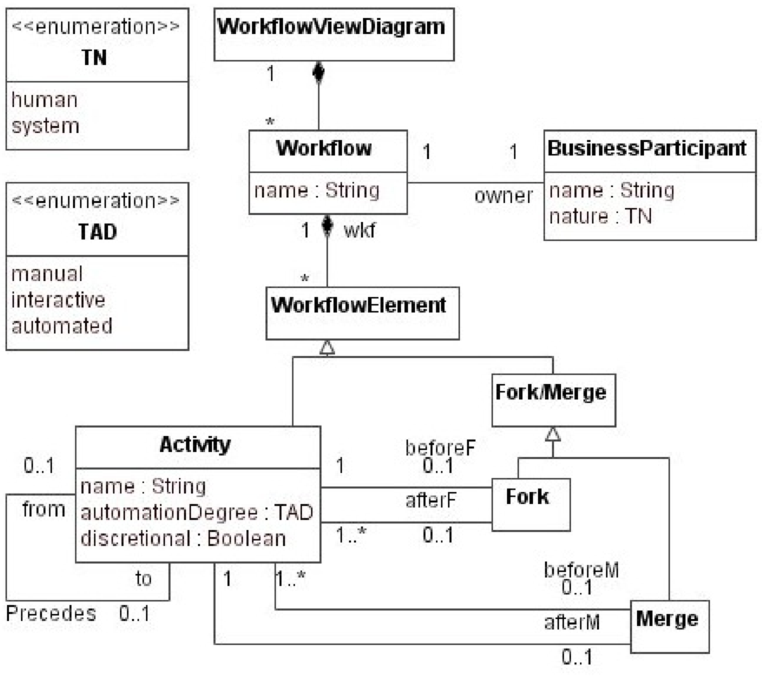
Measure	Patterns applied	Different patterns applied	Expressions not covered by patterns (*)	Measures reused
betweenessCentrality	5	3	0+2+5	0
indegreeCentrality	2	1	0	0
outdegreeCentrality	2	1	0	0
degreeCentrality	2	1	0+0+1	2
closenessCentrality	2	2	0+0+2	0

(*) x+y+z means "x navigations, y operators on collections, z operators not on collections". Navigations include oclAsType (considered as "navigation" in a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators.

BPM measures

SOURCE: Balasubramanian, S., Gupta, M.: “Structural Metrics for Goal Based Business Process Design and Evaluation”. *Business Process Management Journal*, 11(6), 2005.

FRAGMENT OF THE BPM METASCHEMA



METAMODEL CORRESPONDENCE:

BPM diagram	Diagram
WorkflowViewDiagram	Node, Diagram
Workflow	Node, Compound
WorkflowElement, Activity, Fork/Merge, Fork, Merge, BusinessParticipant	Node, Simple
beforeF, afterF, beforeM, afterM, Precedes	Edge, Simple
automationDegree, discretional, nature	Property

CATALOGUE OF MEASURES

Measure: AAF

Definition: Proportion of total activities in a process that require system support.

Formalization:

```
  context WorkflowViewDiagram::AAF(): Real
  post: result =
  byCategoryByPropertyNormalizedDiagram('Activity','autom
ationDegree','automated')
    +
  byCategoryByPropertyNormalizedDiagram('Activity','autom
ationDegree','interactive')
```

Measure: APF

Definition: Longest path of activities that must be executed sequentially divided by the total number of activities.

Formalization:

```
  context WorkflowViewDiagram::APF(): Real
  post:  byCategoryCountDiagram('Activity')  =  0  implies
result = 0
  post: byCategoryCountDiagram('Activity') > 0 implies
        result = allPathsMaxDiagram('Activity',
  'Precedence') / byCategoryCountDiagram('Activity')
```

Measure: TDRF

Definition: Proportion of transitions of flow between business participants from system activities to human activities.

Formalization:

```
  context WorkflowViewDiagram::TDRF(): Real
  post:
  result= byCategoryAndPropertyByTargetElementCategoryAnd
  PropertyFilteringDiagram('Activity','nature','system',
   'Activity','nature','human')->
   select(e | e.wkf.owner <> e.to.wfk.owner)->size()
                / Activity.allInstances()->select(e |
  e.wkf.owner <> e.to.wfk.owner)->size()
```

Measure: PDF

Definition: Proportion of activities performed by human participants that are executed using human discretion or judgement.

Formalization:

```
  context WorkflowViewDiagram::PDF(): Real
  post:
  result =
  byCategoryByPropertyNormalizedDiagram('Activity',
  'discretional','true')
```

Measure: BAF

Definition: Proportion of decision activities in a process that do not require human intervention.

Formalization:

```
  context WorkflowViewDiagram::BAF(): Real
  post:
byCategoryByTargetElementCategoryCountDiagram('Activity',
'Fork') = 0 implies
  result = 0
  post:
byCategoryByTargetElementCategoryCountDiagram('Activity',
'Fork') > 0 implies
result=
byCategoryByTargetElementCategoryFilteringDiagram('Activi
ty', 'Fork')->intersection(
byCategoryByPropertyFilteringDiagram('Activity','automati
onDegree','automated'))->size()
          /
byCategoryByTargetElementCategoryCountDiagram('Activity',
'Fork')
```

SUMMARY:

Patterns applied, individual:

byCategoryByPropertyNormalizedDiagram	3
byCategoryCountDiagram	3
allPathsDiagram	1
byCategoryAndPropertyByTargetElementCategoryAndPropertyFilteringDiagram	1
byCategoryByTargetElementCategoryCountDiagram	3
byCategoryByTargetElementCategoryFilteringDiagram	1
byCategoryByPropertyFilteringDiagram	1

Patterns applied, by category:

Condition	
Attribute conditional	6
Structural conditional	5
Property conditional	2
Result	
Absolute	8
Normalized	2
Crossed	
Filtering	3
Input	
Full diagram	13
Subdiagram	
Individual	

Measures complexity:

Measure	Patterns applied	Different patterns applied	Expressions not covered by patterns (*)	Measures reused
AAF	2	1	0+0+1	0
APF	4	2	0+0+5	0
TDRF	1	1	10+3+5	0
PDF	1	1	0	0
BAF	5	3	0+1+4	0

(*) x+y+z means “x navigations, y operators on collections, z operators not on collections”. Navigations include oclAsType (considered as “navigation” in a hierarchy). Operators on collections are forAll, exists, iterator, select, and by the like. Other operators include not only operators with name (size, asSet, etc.) but also arithmetic, boolean and relational operators.