A Metamodelling Approach for the Definition and Reuse of Structural Metrics

Gemma Grau

Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona 1-3, Barcelona E-08034, Spain.
ggrau@lsi.upc.edu

Abstract. Measurement is a key aspect for Information Systems assessment in all its development phases. Different artefacts allow the application of different kinds of metrics. For instance, when the software system is already developed, test metrics such as benchmarks can be used. However, in the initial phases, metrics can only be applied over model artefacts and, then, structural metrics are especially useful because they allow measuring different properties by taking into account the structural elements of the model. In order to explore how these metrics can be defined and reused, in this paper we analyse several structural metrics over different kinds of models and we observe that they share similar characteristics. Based on them, we establish a metamodelling approach, which includes several guidelines for the definition of the structural metrics and their reuse. In order to exemplify the approach we define and reuse two different structural metrics over the *i** framework.

1 Introduction

Measuring is crucial in many different disciplines and Software Engineering is not an exception. There are many quality characteristics that can be measured such as functionality, reliability, usability, efficiency, modularity, effectiveness, or safety, among others. These characteristics are usually measured once the Information System is built, but they can be estimated in earlier phases of the development process. Therefore, metrics can be applied over different artefacts, for instance, graphs representing the code workflow, conceptual models, or activity diagrams.

Metrics evaluation ranges from expert judgment qualification, to complex computational rules. However, as it is remarked in [19], despite formal estimation models have existed for many years, the dominant estimation method is based on expert judgment, which makes metrics evaluation subjective and time-consuming and hampers reuse of metrics. One of the kinds of metrics that is less based on expert judgment is structural metrics. On spite of their use, as far as we know, there is not a precise definition of structural metrics and, so, we may define them as *those metrics that measure software quality characteristics based on some predefined criteria over the structure of a modelled software artefact*. Structural metrics are very suitable for the early phases of the software development process because models play a prominent role during these phases. Thus, there are many approaches that propose

structural metrics over different models, such as Class Diagrams [5], Statechart Diagrams [11], Use Cases [25], Workflow Diagrams [28], and *i** models [18].

Despite the widespread use of structural metrics, it still does not exist a unified manner to define them, there is no general form for a generic adaptation of metrics across models, and their validation is often assumed. In our own experience in defining structural metrics over *i** models [10], [8], [18], we have particularly remarked: 1) the lack of guidelines for defining the metrics; 2) the need of criteria for establishing when expert judgement has to be used and how; 3) the difficulty on validating the metrics; and, 4) the difficulty on reusing existing structural metrics.

Actually, other authors have also remarked that need of addressing the definition and reuse of metrics in a systematic way. For instance, the work presented in [22] discusses several issues related to model metrics, with particular emphasis on metrics for UML models, and identifies three levels of challenges for model metrics: 1) the technical challenge of defining, comparing and reusing metrics over different descriptions of the same software system; 2) the conceptual challenge of defining how to measure metrics from partial descriptions of models, and of the change in metrics between different representations of the software; and, 3) the practical challenge of gathering, comparing and interpreting new and existing metrics.

In order to address these issues we have analysed most of the existing structural metrics over different modelling languages, looking for their commonalities and differences. Based on this analysis, we have observed that all the studied modelling languages present a similar structure, being possible to establish two different metamodels, graph-based and sequence-based, where most of the studied modelling languages and their metrics fit. Using that metamodel approach we have established the guidelines for a question-based procedure that guides the definition of the metrics from scratch. On the other hand, we have observed that if two modelling languages share the same metamodel, it is then possible to reuse the metrics defined on one modelling language to the other and so, we have also defined the guidelines for doing it. In order to illustrate our approach, we have defined two metrics over *i** models [29], one defined from scratch for measuring Data Accuracy, and the other one by reusing an existing metric for measuring the COSMIC Functional Size.

The remainder of the paper is organized as follows. In Section 2 we provide an overview of some existing structural metrics. In Section 3 we present our metamodelling framework, which can be used for defining metrics from scratch using the guidelines in Section 4, or for reusing existing metrics using the guidelines in Section 5. Finally, in Section 6 we present validation issues, whilst in Section 7 we end with the conclusion and future work.

2 Overview of Existing Structural Metrics

Structural metrics are applied over different domain models for evaluating different quality attributes. The first software metrics were proposed to evaluate some qualities of the software code related with complexity and reuse, by measuring structural elements of the code such as lines of code or the maximum and mean of nested functions. However, as we are interested in applying metrics at the early phases of the

software development process, we focus on artefacts other than code, mainly: UML specifications; Business Process Models; *i** models; and, Functional Size models. Due to the lack of space, we cannot present all the metrics analysed, or the details of the modelling languages used, therefore, for more information we refer to [14].

- **Metrics over UML specifications.** UML models have been the focus of many different structural metrics for evaluating quality factors such as complexity, modifiability and reusability of the modelled artefacts. Among the existing approaches we remark those metrics applied over Class Diagrams [5], [12], Component Models [13], Use Cases [25], and Statechart Diagrams [11]. In Table 1 we show the different kinds of metrics proposed, stating the evaluated UML model and the properties they measure.
- **Metrics over Business Process Models.** Business Process Modelling is related to the software domain because, nowadays, it usually considers the automation of some of the modelled process by means of an Information System. Business Processes are commonly represented as a set of nodes representing states and edges representing transitions between states. Among the existing proposals, we remark the following, which measure complexity over Process Charts [21]; coupling and cohesion over Workflow Models [28]; and complexity metrics over Process Graphs [4]. These proposals have in common that metrics are reused based on the analogy of their models with models on other fields. However, none of them provide a systematic method for doing it.
- **Metrics over *i** Models.** There are a few proposals of structural metrics over *i** models. Among them we remark the ones defined in the REACT method [10], [18] which count the different elements of Strategic Dependency (SD) models for obtaining different values. The work in [8] evaluates Strategic Rationale (SR) models by analysing the structure of their means-end and task-decompositions. Finally, in [3] the SR model is evaluated for the quality attribute Overall Plan Cost.
- **Metrics measuring the Functional Size.** Functional Size Measurement Methods aims at determining the size of a proposed software system yet to be built based on its requirements. It can be measured over the structure of a specific model, such as the Functional User Requirements [1], but also over other constructs such as UML Class Diagrams [24], or the software model of the process [26].

Table 1. Overview of UML-based Structural Metrics, classified by the evaluated UML model

Model	Metric	Property Measured	Reference
Class Diagrams	Weighted Method per Class (WMC)	Complexity (effort)	[5]
	Depth of Inheritance Tree (DIT)	Complexity (behaviour)	[5]
	Number of Children (NOC)	Reusability	[5]
	Coupling between Object Classes (CBO)	Reusability	[5]
	Response for a Class (RFC)	Complexity (testing)	[5]
	Number of Associations	Maintainability	[12]
	Number of Aggregations	Maintainability	[12]
Use Cases	Number of Dependencies (NOD)	Modifiability	[25]
	Number of Use Case Types	Modifiability	[25]
Component Model	Arguments per Procedure (APP)	Complexity	[13]
	Distinct Arguments Count (DAC)	Complexity	[13]
Statechart diagrams	Number of Activities (NA)	Understandability	[11]
	Number of Transitions	Understandability	[11]

3 Adopting a Metamodelling Approach

From the analysis of related work, we observe that structural metrics are all based on counting or weighting the structural elements that conforms the modelling language over which they are defined. For instance, metrics over UML Class Diagrams are based on the number of associations, the number of attributes and combinations of them. We also observe that the modelling languages used also have a similar structure. More precisely, we distinguish among modelling languages that are analogous to a graph or to a sequence of actions. For instance, in UML Class diagrams the classes can be abstracted to nodes and the relationships to edges; in Workflow Processes the states are nodes and the transitions are the edges; and, in *i** models, the actors are the nodes and the dependencies are the edges.

According to [22], *defining model metrics is a metamodelling activity*. This statement is based on the fact that for interpreting and understanding a metrics exact definition, it is necessary to model the entities being measured, and to define the metrics in terms of this model. Based on the analysis of the existing structural metrics that we have presented, we observe that they share the same concepts and, so, it is possible to apply a metamodelling approach. More precisely, some modelling languages are based on a graph structure [3], [5], [8], [10], [11], [12], [13], [25] and some others are based on a sequence-based structure [1], [4], [18], [21], [26], [28]. Therefore, it is possible to abstract a metamodel of these notations and get a more general view of the defined metrics. We remark that these generic metamodels can be transformed into specific metamodels, by applying the refactorings presented in [27].

Therefore, we can establish the three different layers presented in Fig. 1:

- **First Layer: Generic Modelling Language Metamodel.** At this first layer we found the two modelling languages metamodels identified: one for graph-based modelling languages, and the other for sequence-based modelling languages. However, if other metamodels were identified, our approach could host them. The metamodels can be obtained by applying a generalization process over the metamodels of the modelling languages.
- **Second Layer: Modelling Language Metamodel.** Structural metrics are defined over a certain modelling language, and so, this layer contains the metamodels of these modelling languages. The modelling language metamodels may be different from the generic metamodel in the sense that their modelling elements may have different names and slightly different model restrictions and so, they can be seen a refactoring of the metamodels of the layer above. They can also be considered as an abstraction of the specific domain modelling language.
- **Third Layer: Modelling Language Domain Model.** Each modelling language metamodel can be instantiated into many models, depending on its domain. For instance, an *i** model can be instantiated for the domain of e-business systems or service-oriented systems, where models in the same domain share similar concepts.

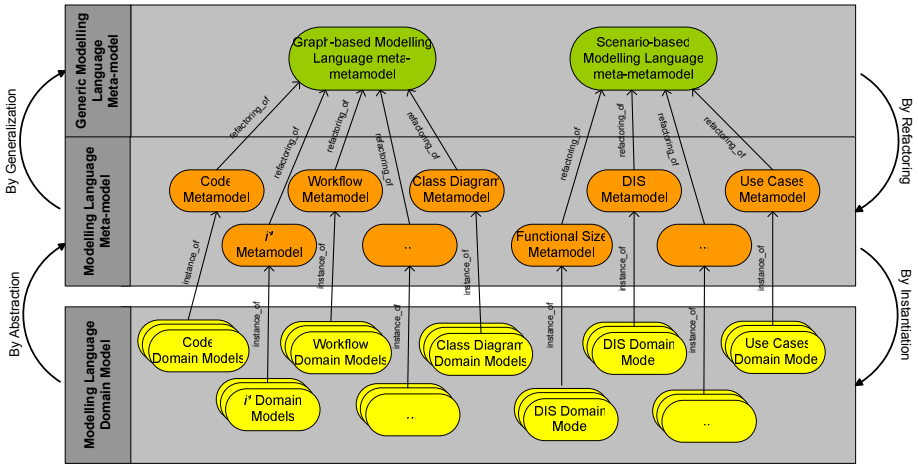

Fig. 1. Structural Metrics modelling languages and the modelling layers proposed

3.1 The Generic Graph-based Metamodel

In Fig. 2, we show our *Generic Graph-based Metamodel*, which is adapted from the *generic metamodel* for gap typology definition presented in [7]. We have selected this metamodel because it is already generic, it is currently being used as a metamodel, and presents the main concepts to be shown in a Graph. The modifications we have done to the original metamodel are: 1) we have renamed the names of the classes Link and Not Link into *Edge* and *Node*, in order to adhere to a generic graph terminology, and, 2) in order to allow a more complete classification of the elements, in addition to the attribute *Name*, we add the attribute *Type* to the class *Element*.

As presented in Fig. 2, an *Element* is classified into two bundles. First, a distinction between *Simple* Element and *Compound* Element is made. Second, elements can be classified into *Nodes* or *Edges*. A *Compound* Element is decomposed into finer-grain elements, which can be *Simple* or, in turn, *Compound* Elements. *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*. For technical reasons, at least one *Element* has to be classified as *Root*. This allows indicating that the minimal content of a model is the Object class in a class hierarchy, the System Boundary in a use case diagram, etc. Finally, an element may have associated one or more *Property*.

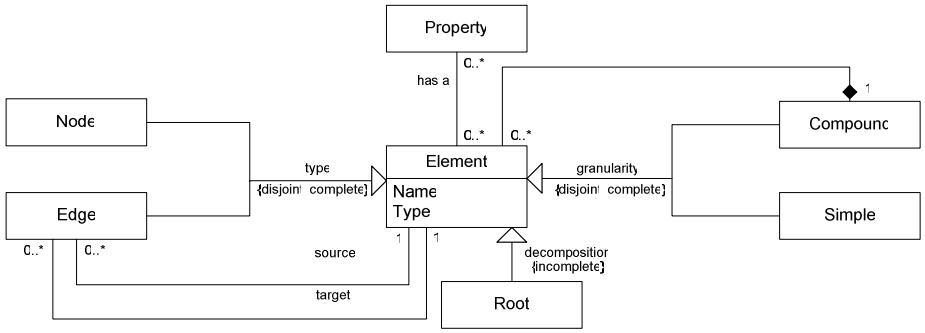

Fig. 2. Generic Graph Metamodel, adapted from [7]

3.2 Generic Sequence-Based Metamodel

In Fig. 3 we present our metamodel for the sequence-based modelling languages (i.e., scenarios, use cases, human activity models, or activity diagrams). In order to adhere to the several existing sequence-based notations, we have define a generic sequence-based metamodel based on the metamodel for Use Cases proposed in [23], and the conceptual model for Use Cases presented in [6]. In order not to enforce a particular notation we have named the classes according to the generic concepts they represent. The main concept is a *Sequence of Actions*, which can be a *Simple Sequence* of Actions or a *Composite Sequence* of Actions (for instance, a Use Case is a set of sequences representing a Scenario [6], and it is also possible to define composite Use Cases [23]). Relationships between *Sequence of Actions* also include *inclusion* and *extension* of other *Sequence of Actions*. *Sequence of Actions* are constrained by a certain number of *Conditions*, which can be a *Goal*, a *Precondition*, a *Postcondition*, or a *Triggering Event* specific of each condition. A *Sequence of Actions* has several *Actions*. An *Action* involves two *Actors* and manages one *Resource*.

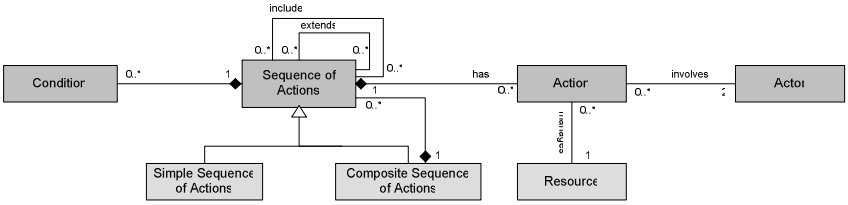

Fig. 3. Generic Sequence-based Metamodel

4 Towards the Definition of Structural Metrics

As we have remarked in Section 2, current approaches using structural metrics do not provide precise guidelines on how to define them. In order to address this issue, we propose to use the two metamodels proposed (graph-based and sequence-based) as a basis for the definition and reuse of structural metrics. In Fig. 4 we present our

process, which involves the following four activities. First, the metamodel of the modelling language is established and, second, based on the metamodel and the general knowledge on existing structural metrics, the structural elements of the modelling language are identified. Regarding the definition of the metrics, they can be defined from scratch by using the metamodel and the structural elements for constructing guidelines that assist its definition. On the other hand, reuse is possible when analogies between both fields can be found [4], [21], [22], [28], and, so, we propose to use the metamodels and the structural elements already identified to define new metrics based on reuse. As shown in Fig. 4, all the generated elements are stored and reused over time.

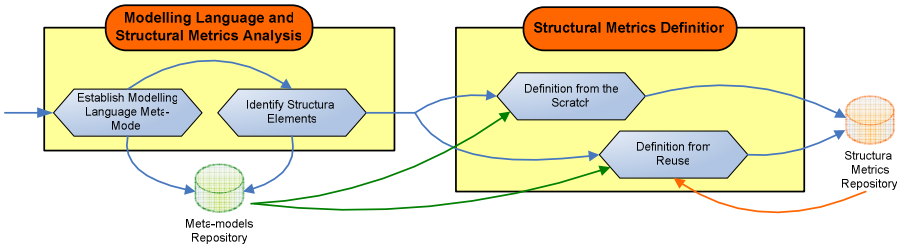

Fig. 4. Process for Defining Structural Metrics

Although it is not mentioned in this paper, we assume that metrics are defined following a metrics definition process such as the GQM approach [2]. The use of a metric definition process ensures that the measurement needs are established, that assumptions are stated before defining the metric, and that metrics are correctly validated and documented.

In order to exemplify our approach in the next sections we define two structural metrics over the *i** framework [29]. There are two kinds of *i** models, each one corresponding to a different abstraction level: the Strategic Dependency (SD) model represents the strategic level by means of the dependencies between the actors, whilst the Strategic Rationale (SR) model represents the rational level by means of showing the intentionality inside each of the represented actors. A SD model consists of a set of nodes that represent actors and a set of dependencies that represent the relationships among them, expressing that an actor (*depender*) depends on some other (*dependee*) in order to obtain some objective (*dependum*). The *dependum* can belong to one of the following four types: goal, task, resource, and softgoal.

There are different modelling techniques for creating *i** models, among which we use the PR*i*M method [18]. In PR*i*M, *i** models are constructed in two steps in order to differentiate the operational process (Operational *i** Model) from the strategic intentionality behind it (Intentional *i** Model). In order to define the Operational *i** Model, PR*i*M uses Detailed Interaction Script (DIS), a sequence-based notation over which it is possible to apply automatic transformation rules in order to obtain the *i** constructs. We remark, then, that *i** models created with PR*i*M may be compliant to both a graph-based metamodel and a sequence-based metamodel. PR*i*M also considers the evaluation of *i** models by means of two kinds of metrics: actor-based and dependency-based. As in our examples we use dependency-based metrics, here

we present their general form. For more details about *i** modelling with PR*i*M and actor-based metrics we refer to [18].

Dependency-based metrics. Given a property P and an *i** SD model M = (A, D), where A is the set of actors and D the dependencies among them, a dependency-based metric for P over M is of the form:

$$P(M) = \frac{\Sigma d:\, d(a,b,x) \in D:\, filter_M(x) \times correctionFactor_M(a,b)}{limit_P(M)}$$

being $filter_M$: D→[0,1] a function that assigns a weight to the every *dependum* (e.g., if the *dependum* is goal, resource, task, softgoal if it is from a specific type), and $correctionFactor_M$: A→[0,1] a function that correct the weight accordingly to the type of actor that the *depender* and the *dependee* are, respectively. This correction factor is often decomposed into: $correctionFactor_{M,der}(a) \times correctionFactor_{M,dee}(b)$, when the *depender* and the *dependee* have a mutually independent influence on the metrics.

4.1 Establish the Modelling Language Metamodel

The first step we propose is to define the metamodel of the modelling language over which the metrics will be defined. We remark that this is done only once for each modelling language. In our example, we want to define metrics over *i** models and so, we establish the metamodel representing the *i** SD constructs. We focus on the part we need for our example. We take into account the *actor* and the *dependum*, as presented in Fig. 5, where the *Actor* is a refactoring of the class *Node* (subtype of *Element*) and the *Dependum* is a refactoring of the class *Edge* (subtype of *i* Element)*. Regarding the Name and Type attributes, an *Actor* can be software (SW), human (H), hardware (HW), or organization (Org); whilst a *Dependum* can be a goal (G), a task (T), a resource (R) or a softgoal (SG). In order to be compliant to the fact that in *i** there is no *dependum* linking an actor with itself, we have to add an integrity constraint. In order to validate the correctness of the defined metamodel, we have checked it against other *i** metamodels such as the one we propose in [9].

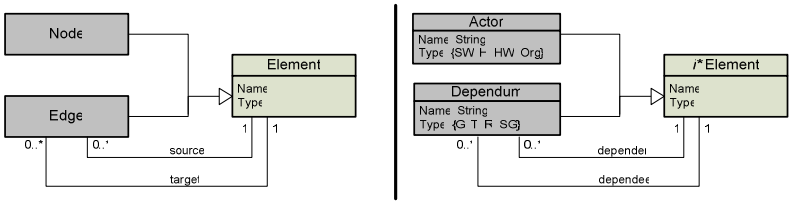

Fig. 5. Excerpt of the Generic Graph Metamodel and its refactoring for *i** SD models

4.2 Identify the Structural Elements

The same analogy that allows establishing a common metamodel among the different modelling languages also provides an analogy on the structural elements used when defining the metrics. In order to facilitate the definition and reuse of the metrics, we

propose to identify common patterns on the use of the structural elements referring on how they can be counted, classified, and weighted when defining the metrics. In [9] we define and document several categories of structural patterns detected in structural metrics over *i** models, we remark:

- **Discrimination Patterns.** We differentiate among *discriminators by type* (Node or Edge according to its particular Type) and *discriminators by name*, which provide a specific value to each element according to a particular characteristic. A typical example of discriminator by type in *i** metrics focus on the type of the *dependum* (Goal, Task, Resource, Softgoal). Concerning discriminators by name, metrics over *i** models often discriminate by *actor name* or by *dependency name*. It is also possible to discriminate a class by other elements such as other attributes (that have to be added to the metamodel), or the existence of relationships with a certain element (e.g., count the number of nodes with a certain value for a given property).
- **Aggregation Patterns.** Given a metric defined over a compound element, they combine the values of the same or another metric applied to its components. Examples are: to count the number of components that satisfy some condition (*count pattern*) or add the values of the component metrics (*sum pattern*)**.** For instance, we have observed that many metrics count the number of elements of a class (number of children [5] in Class Diagrams, or number of states [11] in Statechart Diagrams). On the other hand, the metric *Arguments per procedure* [13] sum the number of arguments for each procedure and then divide by the number of procedures. Additionally, we can also add a discriminator when counting or adding the elements.

In [9] we present other categories of patterns that combine with structural ones, e.g. numerical patterns to normalize metrics' values, implicitly applied in the definition of the metric *Arguments per Procedure* as introduced above.

4.3 Defining Structural Metrics from Scratch

The definition of structural metrics from scratch can be difficult because it is usually done following intuitive procedures and there are no guidelines on how to systematize the process. In order to address this issue, we propose to use the structural elements identified in the previous section to create a set of guidelines for defining structural metrics. These guidelines take the form of a set of questions to be answered in order to customize the metric.

Constructing a questionnaire for *i dependency-based metrics.** In order to construct the guidelines, we analyse the general form of the metric, where we observe that it has three differentiated parts: the $\text{filter}_{M}(x)$, the $\text{correctionFactor}_{M,der}(a)$ and the $\text{correctionFactor}_{M,dee}(b)$. Analysing the structural elements related to the dependencies (see the metamodel in Fig. 5, right), we observe that the structural elements that give a value to the dependency according to its name or its kind (i.e. the *Dependum* node in the metamodel) correspond to the filter and the ones related with the actors that participate in the dependency correspond to the two correction factors (the relationships with the two actor nodes in the metamodel). It is possible to evaluate the correction factors by giving each one a value if they are independent (i.e., defining the

correctionFactor$_{M,der}$(a) and the correctionFactor$_{M,dee}$(b)), or by giving a value to a specific combination of the two (i.e., defining the correctionFactor$_M$(a,b)).

Taking this classification as a starting point, we define the questionnaire presented in Table 2, which states a set of questions that help to identify the elements of each category. We remark that this first classification can be completed with a deeper analysis of the metamodel, but this first level has been sufficient for defining the metrics used in this paper. For defining the table we take into account:

- **filter$_M$(x).** As dependencies are represented as a *Dependum* class and its relationships in the metamodel, we apply a discriminator over it. Discriminator is expressed in terms of the attributes of the *Dependum* class: Type and Name. Therefore, in Table 2, for obtaining the value of filter$_M$(x) we ask a question to discriminate if the type of the *dependum* or the name of a specific dependency actor affects the quality attribute. If none of them affects it, the filter has the neutral value 1. It is also possible to discriminate the duplicated *dependums* positively or negatively by multiplying or dividing for the number of times the *dependum* appears.
- **correctionFactor$_M$(a,b).** The *Dependum* class has two relationships that state the *depender* and the *dependee* of the dependency. If the correction factor concerns a combination of both actors we apply a discriminator by type or by name of each of the actors.
- **correctionFactor$_{M,der}$(a) and correctionFactor$_{M,dee}$(b).** If not the combination but the *depender* and/or the *dependee* type or name individually affects the quality attribute, we apply a *discriminator by type* or a *discriminator by name* in order to obtain the correction factor. As actors may be related with other *Dependum* classes, further discriminator of the related dependencies or actors can be applied. As these operations add complexity, they are not included in the example.
- **limit$_p$(M).** It is used to calculate the average and to normalize the result. Depending on the scale of the metric, the limit can be ignored (value 1).

Table 2. Guidelines for quantifying the dependency-based metric

Element	Question	Answer	Example Value
2.1. Dependency-based: filter$_M$(x)			
	Does the type of the *dependum* or the *dependum* itself affect the quality attribute?		
		No	Filter$_M$(x) = 1
		Yes, the type of the *dependum* affects the quality attribute (*discriminator by type*)	Filter$_M$(x) = { w, if x.Type = Goal; x, if x.Type = Resource; y, if x.Type = Task; z, if x.Type = Softgoal
		Yes, the name of the *dependum* affects the quality attribute (*discriminator by name*)	Filter$_M$(x) = { m, if x.Name = Dep_A; n, if x.Name = Dep_B; *idem* for other names
	Do the duplicated *dependums* affect the quality attribute?		
		Yes, the number of dependencies affects the quality attribute (positively)	Filter$_M$(x) = #Dep(x)
		Yes, the number of dependencies affects the quality attribute (negatively)	Filter$_M$(x) = $\frac{1}{\#Dep(x)}$

2.2. Dependency-based: correctionFactor$_M$(a,b)			
	Does a certain combination of *depender* and *dependee* affect the quality attribute?		
		No	CorrectionFactor$_M$(a,b) = 1
		Yes, a certain *depender* type and *dependee* type affects the quality attribute	Correction Fact$_M$(a,b) = { x, if a.Type = SW ∧ b.Type = H; y, if a.Type = SW ∧ b.Type = SW; *idem* for other combinations
		Yes, a certain *depender* name and *depender* name affects the quality attribute	Correction Fact$_M$(a,b) = { x, if a.Name = Act_A ∧ b.Name = Act_B; y, if a.Name = Act_C ∧ b.Name = Act_D; …
2.3. Dependency-based: correctionFactor$_{M,dee}$(a) - correctionFactor$_{M,dee}$(b) is analogous			
	Does the related *depender* affect the quality attribute?		
		No	CorrectionFactor$_{M,der}$(a) = 1
		Yes, the *depender* type affects the QA	Actor *discriminator by type*
		Yes, the *depender* name affects the QA	Actor *discriminator by name*
2.5. Dependency-based: limit$_p$(M)			
	Which is the scale of the metric?		
		Absolute between [0, infinite]	limit$_p$(M) = 1
		Absolute, average	limit$_p$(M) = ‖ D ‖
		Ratio: [0..1]	limit$_p$(M) = Normalization value

Defining a Data Accuracy structural metric over *i models.** In order to exemplify the application of the guidelines, we define an *i** structural metric for Data Accuracy. Data Accuracy is a quality attribute that measures the degree to which information sources are free from mistakes and errors. From this definition, we observe that data is related with information and so, it is concerned with the resources dependencies of the *i** model. Because of that, we decide that a dependency-based metric is needed.

Before defining the metric, we need to establish certain factors in order to narrow its definition such as its scale and the assumptions over which the measure is done (see Table 3). Based on the assumptions we state that the higher the value of the metric is, the more accurate data is kept in the process. Then, we formalize the metric by applying one guideline of Table 2 for each of the factors of the dependency-based metric as follows (see resulting formalization in Table 3):

- **filter$_M$(x).** Data accuracy depends on the kind of the data, and so, we assume that only resource and task dependencies are important from the information point of view. Data accuracy also depends on the *dependum* name, because each particular data provides a different degree of accuracy on the process. Therefore, we weight the *dependums* according to their accuracy needs. As we are not working with a specific *i** model it is not possible to establish a particular weighting, therefore we propose to classify the *dependums* according to four categories: critical, high, medium, and low. Each category generates a function over dependencies, e.g. *high_accuracy*(x) yields true if x's accuracy is high. In order to weight the filters and correction factors of the metrics, expert advice is strongly recommended.
- **correctionFactor$_{M,der}$(a).** The assumptions state that some actors provide less accuracy than others do. However, when the actor is a *depender*, it is only receiving the data and thus, it cannot introduce mistakes and errors on it unless it becomes *dependee* of the same data. Therefore, the *depender* does not affect data accuracy and we state its neutral value to 1.

Table 3. Documentation of the Data Accuracy *i** metric

General Information	
Metric name:	Data Accuracy
Definition:	Data accuracy is a criterion used in evaluating the quality of information that measures the degree to which information sources are free from mistakes and errors
Scale:	Ratio: [0..1]
Addresses:	Accuracy, Reliability, Fault Tolerance
Source:	Defined from scratch
Metamodel:	*i** metamodel
Assumptions:	- Data accuracy of an *i** model with no dependencies between its actors is 1. - Data accuracy depends on the kind of data being manipulated, as for the correct achievement of the process, some data has to be highly accurate whilst other is not necessary. - Data accuracy depends on the actors that manipulate the data, being human and organizational actors less accurate than software actors.
Formalization:	Dependency-based metric: $DA(M) = \frac{\Sigma d{:}\ d(a, b, x) \in D{:}\ filter_M(x) \times corrFactor_{M,der}(a) \times corrFactor_{M,dee}(b)}{limit_P(M)}$ Where, $filter_M(x) = \begin{cases} 0.2, \text{ if } critical\_accuracy(x) \\ 0.5, \text{ if } high\_accuracy(x) \\ 0.8, \text{ if } medium\_accuracy(x) \\ 1, \text{ if } low\_accuracy(x) \\ 1, \text{ otherwise} \end{cases}$ $correctionFactor_{M,der}(a) = 1$ $correctionFactor_{M,dee}(b) = \begin{cases} 0.7 \text{ if } b.Type = Human \\ 0.7 \text{ if } b.Type = Org \\ 0.9 \text{ if } b.Type = Sw \\ 1, \text{ otherwise} \end{cases}$ $limit_p(M) = \| D \|$
Interpretation:	The higher Data Accuracy value is, the more accuracy, reliability and fault tolerance is provided in the process.

- **correctionFactor$_{M,dee}$(b).** On the other hand, the related *dependee* affects data accuracy because depending on its kind we can consider some actors to be more accurate than others. For instance, if we assume that software systems are correctly built and maintained, they are less prone to introduce mistakes and errors than humans are. Therefore, we weight the *dependees* according to their level of accuracy.
- **limit$_p$(M).** Finally, as the scale of the metric is ratio, we apply a normalization value, which can be the total amount of dependencies on the model.

4.4 Reusing Structural Metrics

There are structural metrics defined over different modelling languages that, due to the analogy on their metamodels, can be reused from one modelling language to another. Based on this analogy, we may think that general rules for transforming the structural metrics from one model to another can be defined. However, as [22] recognizes, to define a generic approach is not that easy because models can vary from each other and so, we need to examine the possibility of mapping the metrics definitions across different models. Because of that, we propose a manual mapping across the metamodels, which is preceded by the selection of the most appropriate

metric. In order to illustrate how reuse is done, we propose to evaluate functional size over *i** models.

Selecting a Functional Size metric. Functional size measures the size of a future software system from the specification of its functional requirements. As *i** models represent both functional and non-functional requirements, they are adequate for measuring the functional size. As the functional size measures the different number of inputs and outputs of the system, we make the assumptions: 1) the functional size of an *i** without software system, is 0; and 2) the more functional dependencies steaming or going through the software system actors are, the higher is the functional size.

Based on these assumptions, among the different functional size metrics, we have selected COSMIC-FFP [1]. In COSMIC-FFP, the functional size is estimated based on the Functional User Requirements specification of software systems, which distinguishes three types of actors: Functional User (FU), Functional Process (FP), and Persistent Storage (PS). Depending on the actors involved, it distinguishes four types of data movement: an Entry moves a data group into the FP from a FU; an Exit moves a data group out of the FP to a FU; a Write moves a data group from the FP to a PS; and, a Read moves a data group from a PS to a FP. The COSMIC-FFP functional size is calculated by assigning to each data movement, a single unit of measure which is, by convention, equal to 1 CFU (Cosmic Functional Unit). The total size of the software being measured corresponds, therefore to the addition of all data movements as follows:

$$\text{Size (functional process)} = \Sigma \text{ size}(\text{Entries}_i) + \Sigma \text{ size } (\text{Exits}_i) + \Sigma \text{ size } (\text{Reads}_i) + \Sigma \text{ size } (\text{Writes}_i)$$

Mapping of the metamodelling concepts. Once a suitable metric is found, we have to check if the mapping between the two metamodels is possible and if there is some equivalence between the identified structural elements. Regarding the measurement of COSMIC over *i**, the COSMIC Functional User Specification metamodel is sequence-based and the operational *i** model defined with PR*i*M (DIS diagrams, see Section 4) is also sequence-based. On the other hand, the structural elements identified in the COSMIC metric, follow the *discrimination patterns* and the *aggregation patterns*, which are also used in *i** structural metrics. This check ensures a correct mapping of concepts, which is done following the process in Fig. 6.

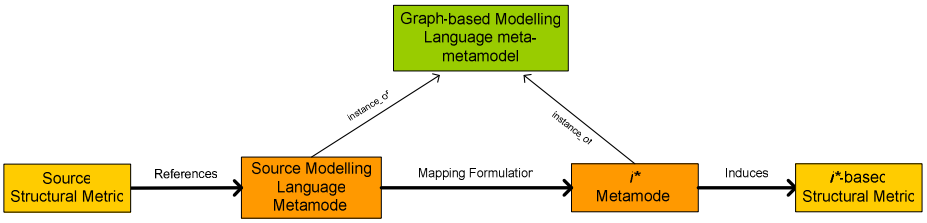

Fig. 6. Structural Metrics reuse process based on metamodels

Based on this reuse process, in Fig. 7, we establish the set of mapping relationships between the concepts needed in the Functional User Requirements (FUR) of the COSMIC generic software model, the information documented in the DIS tables and the *i** concepts. At the left of Fig. 7, we observe that the COSMIC method is based

upon a *Functional Process* which has a *Triggering Event* and several *Subprocesses* associated to it. Each *Subprocess* has a Data Group that can be of the type: entry (E), exit (X), read (R) or write (W). In the DIS, each *Functional Process* is represented by an *Activity*; the *Triggering Event* is part of the *Conditions* associated to the *Activity*; and, each *Subprocess* is represented by the concept of an *Action*. There is a correspondence between the concepts of *Data Group* and *Resource*, although the distinction between the *Data Group* types is implicit in the DIS information because it depends on the *Actors* that participate in the action. As we have already mentioned, [18] proposes a set of automatic rules to transform DIS into *i** Models, where *Conditions* are transformed into *Goal Dependencies*; *Activities* and *Actions* are represented into SR elements; and, *Resource Dependencies* are established between the different *Actors*. In order to help the evaluation of the *i** Model with the COSMIC method, we have added an instance of the Property metaclass (see the graph-based metamodel in Fig. 2), "Type of Cosmic Actor", to the *Actor* in order to allow its classification into FU, FP, and PS.

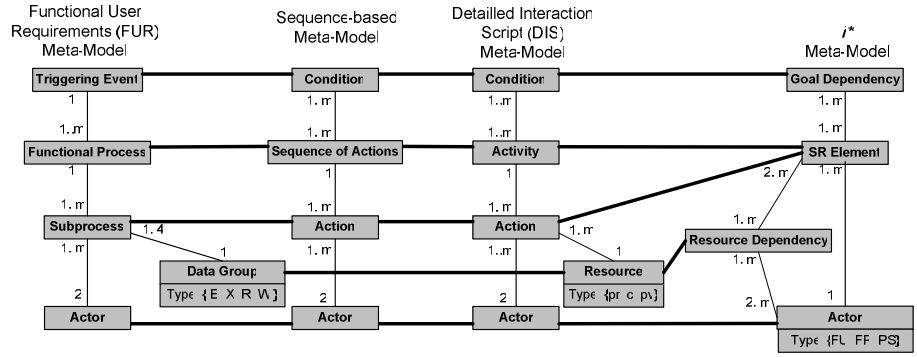

Fig. 7. Reusing the COSMIC metric: Mapping across the different metamodels

Defining the COSMIC metric for *i.** Once the mappings are established, the metric can be defined by establishing the analogous concepts. The metric is documented in Table 4, and these are the criteria stated for its formalization:

- **$\mathbf{filter_M(x)}$.** Data Groups are analogous to resources and, so, it indicates that only resource dependencies are taken into account with a value of 1 (equal to 1 CFU).
- **$\mathbf{correctionFactor_M(a, b)}$**. Since COSMIC only takes into account data movements between certain pairs of actors, we do not split the correction factor into two (one for *depender* and the other for *dependee*). To define the correction factor, we use the "Type of Cosmic Actor" property (shortened as "CosmicType" in Table 4): we assign a value of 1 (equal to 1 CFU) when the dependency accomplishes the data movement restrictions between the predefined actor pairs, and 0 otherwise.
- **$\mathbf{limit_p(M)}$.** Finally, as the scale of the metric is absolute, as the COSMIC formula does not require any normalization, $limit_p(M)$ gets its neutral value, 1.

Finally, the metric has been validated by replicating the case studies provided by the COSMIC method [1], and obtaining the same results (see [15], [17] for details).

Table 4. Documentation of a COSMIC metric over *i** models

Metric name	***i*-based COSMIC Functional Size***
Definition	Metric that measures the Functional Size of a software product based on its *i** model. The unit of measure for the COSMIC functional size is CFS.
Scale	Absolute. The only valid values are zero and positive integers.
Addresses:	The functional size is an indicator for complexity, development effort, and maintainability of the specified software system.
Source:	Adapted from the COSMIC method [1].
Metamodel:	*i** metamodel
Assumptions:	- The functional size of an *i** without software system, is 0. - The more functional dependencies steaming or going through the software system actors, the higher is the functional size.
Formalization:	$Functional\_Size(M) = \frac{\Sigma d:\, d(a, b, x) \in D:\, filter_M(x) \times correctionFactor_M(a,b)}{limit_P(D)}$ Where, $filter_M(x) = \begin{cases} 1, \text{ if } x.\text{Type} = \text{Resource} \\ 0, \text{ otherwise} \end{cases}$ $correctionFactor_M(a,b) = \begin{cases} 1, \text{ if } CosmicType(a) = \text{FP} \wedge CosmicType(b) = \text{FU} \\ 1, \text{ if } CosmicType(a) = \text{FU} \wedge CosmicType(b) = \text{FP} \\ 1, \text{ if } CosmicType(a) = \text{FP} \wedge CosmicType(b) = \text{PS} \\ 1, \text{ if } CosmicType(a) = \text{PS} \wedge CosmicType(b) = \text{FP} \\ 0, \text{ otherwise} \end{cases}$ $limit_P(D) = 1$
Interpretation:	The higher the Functional Size is, the larger the final software system will be.

6 Validation Issues

The presented metamodelling approach for the definition and reuse of structural metrics is used as part of the step for the evaluation of several alternative *i** models in the PR*i*M method [18]. In order to validate PR*i*M, in [14] we applied the method over three formative case studies and one industrial case study. As structural metrics are difficult to apply without tool support, the case studies were done using the tool support provided by J-PR*i*M [16], which supports the different phases of the PR*i*M method.

The formative validation was done using three common exemplars on the software engineering field, namely: the Meeting Scheduler, the Collaborative Exercise, and the Conference Management System. In order to validate the metrics, we defined a comparative hypothesis stating that the evaluation of the alternative *i** models belonging to the different case studies had to be consistent with the properties of the generated alternatives. The properties defined were: Data Accuracy, Data Privacy, Ease of Communication, and Process Agility; and they were defined once for the Meeting Scheduler case study and reused across the other two exemplars. Regarding the stated hypothesis, the evaluation results where constant in the three exemplars, in the way that the obtained values where consistent with the assumptions taken during the definition of the metrics, even if the exemplars where different. Regarding the

time invested, the definition of the four metrics took a total amount of 58 minutes for the Meeting Scheduler case study (using J-PR*i*M); whilst the reuse of the metrics on the other two case studies took less: 35 minutes for the Collaborative exercise case study and 33 minutes for the Conference Management case study. The time of computing the metrics was not taken into account as it was done automatically by the tool, J-PR*i*M. The size of the *i** models where the metrics were defined and applied was less than a 100 structural elements per model.

In the other hand, the proposed approach was also applied over an industrial reengineering case study, doing a *replicated product design* [2], which consists on comparing the results of using a new method against a company baseline. Therefore, the case study was applied on-line by a control team and we replicated the case study off-line in order to compare the results. As it was not possible to know what the other team was doing, the final results could not be compared because they applied a qualitative approach, whilst structural metrics are quantitative. However, we selected the same alternative even using different evaluation criteria. Regarding the time invested, the definition of the structural metrics took a total amount of 4 hours 52 minutes for defining seven structural metrics. Among them, three where reused from the case studies (Data Accuracy, Ease of Communication, and Process Agility) and four where defined from scratch (Average Actor Workload, Data Consistency, Data Truthfulness, and Uniformity of the User Interface). Again the time of computing the metrics is not taken into account as it is done automatically by the tool, J-PR*i*M. The size of the *i** models for the industrial case studies was around 1000 elements per model. We refer to [14] for more details on the definition and execution of the case studies and for the catalogue of the structural metrics defined.

7 Conclusions and Future Work

There is an increasing use of structural metrics, and also, an increasing need of the reuse of existing metrics. However, most of the proposed structural metrics are defined in an intuitive way, without applying any metric definition process. In order to facilitate metrics definition and reuse, we have defined a set of systematic guidelines for: 1) defining metrics from scratch based on the structural elements of the modelling language and based on the knowledge of the existing ones; and, 2) facilitating metrics reuse based on applying a mapping process across the metamodels of the modelling languages over which the metrics are applied. The guidelines have been applied to define two metrics over *i** models for evaluating Data Accuracy and the COSMIC Functional Size.

Regarding validation, the guidelines for the reuse of metrics are based upon the observation of existing structural metrics and, so, the detected structural factors are based on the existing structural metrics which provides a structural validity. However, metrics defined from scratch have to be validated in order to ensure reliability in the results, which can be done by applying techniques such as the ones used in [2], [24]. In our work, structural metrics are validated by applying them in different case studies and, then, checking that the results verify the stated the assumptions and provides the expected results (see [14] for more details). On the other hand, we remark that the *i**

metrics that are defined from reuse, benefit from the validation of the source reused metric. For instance, it is also possible to replicate the experiments or case studies on which the reused metric is applied, such as we have done in [15], [17].

Based on the results obtained, we provide different solutions for the use of structural metrics. First, we propose a set of guidelines that address the construction of a questionnaire for defining the metrics from scratch, including the statement of a certain criteria for applying expert judgement when needed. Second, in order to overcome the difficulty on validating the metrics, we propose guidelines for reusing existing structural metrics and, thus, benefit from previous validation. Therefore, we observe that reuse of structural metrics is very likely because they are all based on the structural elements of a modelling language that it is usually represented in a graph-based or sequence-based structure. This last point particularly addresses the issues raised in [22] as our approach allows: 1) defining, comparing and reusing metrics over different descriptions of the same software system; 2) defining how to measure metrics from partial descriptions of models, 3) changing the metrics between different representations of the software; and 4) the practical challenge of gathering, comparing and interpreting new and existing metrics.

As a future work, we aim at exploring the use and reuse of structural metrics over *i** models by defining a complete catalogue of structural metrics and of structural patterns [9]. We will also define a collection of generic metrics and patterns over the metamodels, and explore the possibility of doing it over the MOF. Tool support will also be developed in order to improve the applicability of the approach.

Acknowledgements. The author wants to thank the ER 2008 anonymous reviewers for their comments and suggestions on the earlier version of this paper. The author also thanks Xavier Franch for his valuable suggestions. This work has been supported by an UPC research scholarship.

References

1. Abran, A., et al: "COSMIC Method Version 3.0, Measurement Manual". The Common Software Measurement International Consortium, 2007. Available at: http://www.gelog.etsmtl.ca/cosmic-ffp/COSMIC-MethodV3.html.
2. Briand, L.S., Morasca, S., Basili, V.R.: "An Operational Process for Goal-Driven Definition of Measures". IEEE Trans. Software Eng. 28(12): 1106-1125 (2002).
3. Bryl, V., Giorgini, P., Mylopoulos, J.: "Designing Cooperative IS: Exploring and Evaluating Alternatives". OTM Conferences (1) 2006. pp. 533-550.
4. Cardoso, J., Mendling, J., Neuman, G., Reijers, H.A.: "A Discourse on Complexity of Process Models". In *Proceedings of BPM 2006 Workshops*. LNCS 4103. pp. 117-128.
5. Chidamber, S.R., Kemerer, C.F.: "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering. Vol. 20, No. 6., June 1994.
6. Cockburn, A.: *Writting Effective Use Cases*. Addison-Wesley, 2000
7. Etien, A., Deneckère, R., Salinesi, C.: "Extending Methods to Express Change Requirements". In *Proceedings of EMSISE 2003*.
8. Franch, X.: "On the Quantitative Analysis of Agent-Oriented Models". In *Proceedings CAiSE 2006*. Springer-Verlag, LNCS 4001, pp. 495-509
9. Franch, X., Grau, G.: "Towards a Catalogue of Patterns for Defining Metrics over *i**

Models". To appear in *Proceedings of CAiSE 2008*. Available for reviewing purposes at: www.lsi.upc.edu/~ggrau/er08/caise2008.pdf
10. Franch, X., Maiden, N.A.M.: "Modelling Component Dependencies to Inform their Selection", In *Proceedings ICCBSS 2003*. Springer-Verlag, LNCS 2580. pp. 81-91
11. Genero, M., Miranda, D., Piattini, M.: "Defining and Validating Metrics for UML Statechart Diagrams". In *Proceedings of QAOOSE 2002*.
12. Genero, M., Piattini, M., Calero, C.: "Empirical Validation of Class Diagram Metrics". In *Proceedings of ISESE 2002*.
13. Goulao, M., Brito e Abreu, F.: "Formal Definition of Metrics Upon the CORBA Component Model". In *Proceedings of QoSA-SOQUA 2005*, LNCS 3712, pp. 88-105.
14. Grau, G.: *An i*-based Reengineering Framework for Requirements Engineering*. PhD. thesis, Universitat Politècnica de Catalunya, 2008.
15. Grau, G.: "Adapting the COSMIC Method for Evaluating the Functional Size in PR*i*M ". In *Software Process and Product Measurement*. Springer-Verlag, LNCS 4895, pp: 139-153, 2008.
16. Grau, G., Franch, X., Ávila, S. "J-PR*i*M: A Java Tool for a Process Reengineering *i** Methodology". In *Proceedings of RE 2006*. pp. 352-353.
17. Grau, G., Franch, X.: "Using the PR*i*M method to Evaluate Requirements Models with COSMIC-FFP". In *Proceedings of IWSM-MENSURA 2007*. pp. 110-120.
18. Grau, G., Franch, X., Maiden, N.A.M. "PR*i*M: an *i**-based process reengineering method for Information Systems specification". Information and Software Technology. Volume 50, Issue 1-2, pp. 76-100 (2008)
19. Jorgensen, M., Shepperd, M.: "A Systematic Review of Software Development Cost Estimation Studies". IEEE Transactions on Software Engineering, Vol. 33, No. 1, January 2007.
20. Kitchenham, B., Pickard, L., Pfleeger, S.L.: "Case studies for method and tool evaluation". IEEE Software. Vol. 12, Issue 4. July, 1995.
21. Latva-Koivisto, A.M.: "Finding a complexity measure for business process models". Research Report. Helsinki University of Technology, Systems Analysis Laboratory, 2001.
22. McQuillan, J.A., Jacqueline, Power, J.F.: "On the Application of Software Metrics to UML Models". In *Proceedings of MoDELS 2006 Workshops*, LNCS 4364, pp. 217-226, 2006.
23. Nakatani, T., Urai, T., Ohmura, S., Tamai, T.: "A Requirements Description Metamodel for Use Cases". In *Proceedings of APSEC 2001*. pp: 251-258.
24. Poels, G.: "Definition and Validation of a COSMIC-FFP Functional Size Measure for Object-Oriented Systems". In *Proceedings of QAOOSE 2003*.
25. Saeki, M.: "Embedding Metrics into Information Systems Development Methods: An Application of Method Engineering Technique". In *Proceedings of CAiSE 2003*, LNCS 2681, pp. 374-389, 2003.
26. Santillo, L., Conte, M., Meli, R.: "Early & Quick Function Point: Sizing More with Less". In *Proceedings of METRICS 2005*.
27. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.M.: "Refactoring UML Models". *Proceedings of UML 2001*, pp. 134-148.
28. Vanderfeesten, I., Cardoso, J., Mendling, J., Reijers, H.A., van der Aalst, W.: "Quality Metrics for Business Process Models". BPM and Workflow Handbook 2007. Future Strategies Inc., Lighthouse Point, Florida, USA, 2007, pp. 179-190.
29. Yu, E.: *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995.