Contextual Reusability Metrics for Event-Based Architectures

Sutirtha Bhattacharya
PTD Automation
Intel Corporation
Hillsboro, OR – 97124
sutirtha.bhattacharya@intel.com

Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
Austin, TX 78712
perry@ece.utexas.edu

Abstract

Component Based Software Engineering has been perceived to have immense reuse potential. This area has evoked wide interest and has led to considerable investment in research and development effort. Most of these investigations have explored internal characteristics of software components such as correctness, reliability, maintainability, modularity, understandability, readability, interoperability, portability, generality and genericity for promoting reuse. But experience over the past decade has demonstrated that the usefulness of a component depends as much on the context into which it fits as it does on the internal characteristics of the component. This context takes into account the requirements of the domain and an architectural description is a useful way of representing that domain. In this paper, we present a set of reusability metrics designed to measure how well a software component fits into such an architectural context.

Keywords

Component Based Software Engineering, Reuse, Metrics, Architectural Description

1. Introduction

Almost a decade and a half of architectural research, beginning with the Perry and Wolf paper [1], has resulted in significant progress in the area of Software Architecture, but it is evident that software engineering is still far from the maturity of other traditional engineering disciplines. Software Architecture was envisioned to be the agent that would catalyze the transformation of software engineering into a well understood discipline by driving standardization, developing architecture templates for well-understood domains and enabling systematic reuse of architectural components. That clearly has not happened. However, progress in the areas of Model Driven Architectures, Product Line Architectures, Architecture Description Languages and Architecural Styles form a strong basis and motivation for reuse.

Software Reuse research has seen significant activity over the years. To quantify the benefits of reuse and for supporting objective descision making, reuse metrics have long been a subject of interest. It has been widely felt that in some sense, researchers have fully explored most of the traditional methods of measuring reusability: complexity, module size, interface characteristics, etc. Though the research community does currently recognize the importance of the problem domain with regard to reuse, few have actually linked the context in which a component is used to the true "usefulness" of that component. We believe reuse research will benefit greatly by focusing on the framework in which a component fits. So, if the reusability of a component depends on context, then reusability metrics need to include characteristics about the domain, the software architecture, and the associated environment.

This paper discusses the use the software architecture descriptions as the 'context' of a software component. Our contextual metrics enable quantitative evaluation of the reusability of a software component based on its compliance to different elements of an architecture description. Reuse evaluations are also promoted by using these metrics to quantitatively evaluate the similarity between different components, measure a component's coverage of functionality encoded in the architectural description and numerically track the evolution of a component in terms of system data and functionality.

Section 2 of this paper gives a brief outline of related work done in the area of reusability measurement. Section 3 discusses the assumptions about the architectural 'context'. The proposed metrics are elaborated in Section 4. Section 5 briefly explains the use of these metrics and Section 6 concludes the paper.

2. Reuse Metric Approaches

State of the art approaches for measuring reusability fall into two basic categories: empirical and qualitative. Empirical methods depend on objective data and can normally be calculated automatically and inexpensively [9] while the qualitative methods generally rely on subjective assessment of the software's adherence to some guidelines or standards [9]. We draw from Jeff Poulin's book [5] to navigate these spaces of reusability measurement.

Empirical Methods: One of the most prominent approaches in this area is that by Prieto-Diaz and Freeman. They identified several program attributes - program size, program structure, program documentation, and reuse experience and proposed a faceted classification scheme for evaluating reusability based on these attributes [2]. Selby, on the other hand, proposed a module oriented, statistical study of reusability characteristics of software [3]. The ESPIRIT-2 project called REBOOT (Reuse Based on Object Oriented Techniques) developed a taxonomy of reusability attributes with four reusability factors[4]. Caldiera and Basili [6] proposed a module oriented empirical approach in which they stated that basic reusability attributes depend on component costs, quality, and usefulness. Using ideas drawn from plagiarism detection, Hislop proposed a module-oriented approach for evaluating components in terms of function, form and similarity [7]. Boetticher and Eichmann [8] explored the viability of using neural networks to generate reusability rankings of software. Torres and Samadzadeh established a relationship between information theory metrics and reusability metrics and concluded that reuse information metrics might help in selecting the optimum case among different reuse candidates [10].

Qualitative Methods: Since defining an objective reusability metric often proves difficult, many organizations provide subjective (non-empirical) guidance on identifying and building reusable software components. Some of the prominent approaches in this area include Edwards [11], Hooper and Chester [12], Hollingsworth [13] and NATO [14]. These guidelines generally involve an intuitive description of what a reusable component ought to look like and range in content from general discussions about designing for reuse to rigorous design points [13, 15]. Usually module oriented, the guidelines often elaborate on formatting and style requirements and identifies general "reusability" attributes. Notable among the studies on "reusability" attributes is the work of Khairuddin and Key, who have examined these attributes to construct a reusability model [16]. Another notable approach, the "3C Model," [17] attempts to isolate the three design point specific dependencies of concept, content and context from each other during the implementation and design of a module.

Summary: With the exception of the 3C Model, none of the approaches mentioned above include any software architecture or domain characteristics. They typically explore a component's internal characteristics, which do not take into account the context (the requirements and architectural structure) in which the software component operates.

The set of metrics presented here quantitatively evaluates a software component with respect to (1) compliance/adherence to those functional and data requirements captured in the architectural description, (2) compliance/adherence to the architecture structure, (3) the architecture compliance and coverage of the domain architectural descriptions and (4) the evolution of compliance and coverage over different component versions. These quantitative, contextual evaluations position this research as fundamentally different from previous work done in this area.

3. Context Assumptions

The *context* of a software component is encoded in some form of system description. In 1980, Perry and Habermann [22] proposed a system description language and identified the rules for well-formed system compositions in terms of required and provided elements in configuration compositions. These compositions defined the context for evaluating the substitutability (or reuse) of one component for another. Since then we have seen the advent of architecture description languages (ADLs) to define basic system structures and establish constraints on those structures, their individual components and component interactions.

In this research we use architecture descriptions to define the *context* for use and reuse. Further, we make the following basic assumptions about these *context* descriptions in terms of their descriptive elements and format. The description of each component in a architecture description consists of at least the following

- Interface descriptions of the services that include associated input and output descriptions of the data and events, and the pre & post conditions;
- Attributes descriptions; and
- Behavioral descriptions.

Interface descriptions of services are universally standard in almost all architecture description languages. Pre and Post conditions have been used in several formal approaches to architectural description e.g. Inscape [23].

Event based behavioral descriptions have gained in popularity with Luckham's Rapide [24].

With the above basis for the context, this research can be extended to model driven architectures, product line architectures, reference architectures and different expressions of architectural styles with less complete descriptions.

On the assumption that we have an asset base from which we choose components to use in the architecture description to instantiate that architecture, we propose the model in Figure 1 of an asset component specification to capture the necessary information to be used in the contextual metrics. We note that it consists of the same information we assume to be present in the architecture description.

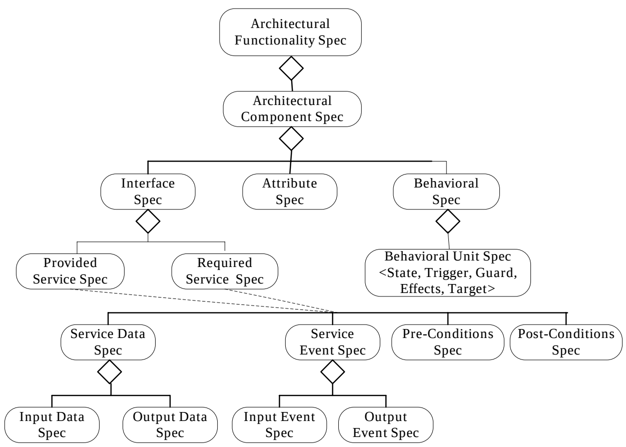

Figure 1: Asset Component Specification

Creating the asset base then requires a specification activity, referred to here as *Registration*, to establish the necessary information needed to measure the usability of a component in a particular architecture.

The following steps detail the Registration process:

Step 1: Select an architectural component from the architectural specification

Step 2a: For the services provided by the architectural component, capture the services *supported* by the component being specified in the Provided Service Specification.

For each service:

- Capture the Input and Output Data & Events supported by the component for the service in the Data and Event Specification
- Capture the Pre and Post Conditions supported by the component for the provided service in the Pre and Post Condition Specifications

Step 2b: Capture the services *required* by the component in the Required Services Specification by following the steps similar to Step 2a

Step 3: For the Attributes for the architectural component, capture the attributes supported by the component in the Attribute Specification

Step 4: For the Behavioral Units of the architectural component, capture the behavioral units supported by component in the Behavioral Unit Specification in the form of quintuples (State, Trigger, Guard, Effects and Target). A component may be registered to a subset of the quintuples for each state transition.

Step 5: Repeat Steps 1-4 for all architectural components in the architectural specification.

4. Proposed Metrics

In order for a software component to be reusable, its engineering characteristics need to be compatible with the envisioned target system in terms of its functional requirements. Though it may be possible to institute some well-defined approaches to engineer components that ensure reusability, there is still considerable amount of debate on this issue [18]. The set of metrics presented in this research, support a quantitative and objective evaluation of software components with respect to the context of the architecture description --- (i) the domain (functional and data) requirements contained in the architecture description and (ii) the architectural components.

The metrics are categorized into Architecture Compliance Metrics and Component Characteristic Metrics.

4.1 Architecture Compliance Metrics

The Architecture Compliance Metrics measure the compliance of a software component to the constituent elements of the architecture description at different levels of granularity. The key metric is the *Architectural Component Compliance Coefficient* which measures the compliance of an asset component to a particular component in the architecture description, taking into account the (i) interfaces supported and required (ii) the data owned and (iii) behavior. For computing this metric, three other metrics are relevant – the *Architectural Component Service Compliance Coefficient*, the *Architectural Component Attribute Compliance Coefficient* and the *Architectural Component Behavior Compliance Coefficient*. This intuitively is analogous to our representation, thus providing objective measures for the three key aspects of any software component – the interfaces, the data, and the behavior. These compliance coefficients can be used to compare different software components for identifying a "best-fit" candidate when designing a system and attempting to reuse previously developed components.

The metrics are discussed below.

4.1.1 Architectural Component Service Compliance Coefficient.

The Architectural Component Service Compliance Coefficient, *ArchSvCoeff(d),* is a measure of an asset component's compliance to all the provided as well as the required services of a particular architectural component. It is computed as the average of the asset component's compliance to each of the required architectural services

Service level compliance indicates the extent to which an asset component is compliant with a given functional requirement (service) defined in the architectural description. Six coefficients, namely the Input Data Compliance Coefficient, the Output Data Compliance Coefficient, the Input Event Compliance Coefficient, the Output Event Compliance Coefficient, the Pre-Conditions Compliance Coefficient and the Post-Conditions Compliance Coefficient are used to calculate the service level compliance. Therefore for each service, *s*, in an architectural component, the following coefficients are defined:

Input and Output Data Compliance Coefficient

The Input/Output Data Compliance Coefficients, *IDCoeff(s)/ ODCoeff(s),* are measures of how well an asset component registered to a given service s, in the architectural component, complies with the input /output data requirements for that service. These coefficients are essentially the average of the ratios between the number of data elements to which the component is registered to the total number of data elements associated with a particular data entity, for all the input/output data entities. A data entity can be thought of as a data concept like 'Address', while data elements are the sub-elements of that data concept like Street Address, City, State, Zip for our example. Thus

$$IDCoeff(s) = \frac{1}{|IDEn(s)|} \sum_{en \in IDEn(s)} \frac{|IDEl_{regd}(s,en)|}{|IDEl(s,en)|} \quad (1)$$

$$ODCoeff(s) = \frac{1}{|ODEn(s)|} \sum_{en \in ODEn(s)} \frac{|ODEl_{regd}(s,en)|}{|ODEl(s,en)|} \quad (2)$$

Where

IDEn(s)/ ODEn(s): Set of Input/Output Data Entities for service *s*.

$IDEl_{regd}(s, en)/ ODEl_{regd}(s, en)$: Set of Input/Output Data Elements for the entity *en* of service *s*, to which the component is registered.

IDEl(s, en)/ODEl(s, en): Set of Input/Output Data Elements for the entity *en* of service *s*.

en: An entity belonging to the set *IDEn(s)/ODEn(s)*

Input and Output Event Compliance Coefficient:

The Input/Output Event Compliance Coefficient, *IECoeff(s)/OECoeff(s),* measures an asset component's compliance to the input/output event requirements of a component service, s, in the architectural description. *IECoeff(s)/ OECoeff(s),* are the ratios between the total number of input/output events to which a component is registered to the total number of input/output events for service s.

$$IECoeff(s) = \frac{|IE_{regd}(s)|}{|IE(s)|} \quad (3)$$

$$OECoeff(s) = \frac{|OE_{regd}(s)|}{|OE(s)|} \quad (4)$$

Where

$IE_{regd}(s) / OE_{regd}(s)$: Set of Input/Output Events for service *s*, to which component is registered.

$IE(s) / OE(s)$: Set of Input/Output Events for service *s*.

Pre and Post Condition Compliance Coefficient

The Pre/Post Condition Compliance Coefficient, *PreCondCoeff(s)/PostCondCoeff(s),* measures an asset component's compliance to the pre and post condition requirements of a component service, s, in the architectural description. *PreCondCoeff(s)/ PostCondCoeff(s)* are the ratio between the total number of pre/post conditions to which a component is registered to the total number of pre/post conditions for service s.

$$PreCondCoeff(s) = \frac{|PreCond_{regd}(s)|}{|PreCond(s)|} \quad (5)$$

$$PostCondCoeff(s) = \frac{|PostCond_{regd}(s)|}{|PostCond(s)|} \quad (6)$$

Where

$PreCond_{regd}(s)/PostCond_{regd}(s)$: Set of Pre/Post Conditions for service *s*, to which component is registered.

PreCond(s)/PostCond(s): Set of Pre/Post Conditions for service *s*.

Using the above six coefficients, for a service s, we obtain a value for the compliance of an asset component to the service, s.

Service Compliance Coefficient:

The Service Compliance Coefficient, SvCoeff(s) measures an asset component's overall compliance to the architecture component's service, taking into account its compliance to input and output data (*IDCoeff* & *ODCoeff*), input and output events (*IECoeff* & *OECoeff*) and pre and post conditions (*PreCondCoeff* and *PostCondCoeff*). The Service Compliance Coefficient also takes into account the relative importance of the particular service in the architecture by considering the number of other services that directly affects or is affected by the service under consideration. SvCoeff(s) is essentially the weighted average of the Input and Output Data Compliance Coefficient, the Input and Output Event Compliance coefficient and the Pre and Post Condition Compliance Coefficient.

$$SvCoeff(s) = \frac{\begin{array}{l}[IDDep(s)\times IDCoeff(s) + ODDep(s)\times ODCoeff(s)\\ + IEDep(s)\times IECoeff(s) + OEDep(s)\times OECoeff(s)\\ + PreCondDep(s)\times PreCondCoeff(s)\\ + PostCondDep(s)\times PostCondCoeff(s)]\end{array}}{\begin{array}{l}IDDep(s) + ODDep(s) + IEDep(s) + OEDep(s)\\ + PreCondDep(s) + PostCondDep(s)\end{array}} \quad (7)$$

Where

IDDep(s)/ ODDep(s): Total number of services generating the input/output data entities required by service *s*.

IEDep(s)/OEDep(s): Total number of services that generate/depend on all the trigger events of /from the service s.

PreCondDep(s)/PostCondDep(s): Total number of services responsible for the set of pre/post conditions

The Service Compliance Coefficient for each service is calculated for each service that is *provided* or *required* by the component being registered. Finally, we compute the Architectural Component Service Compliance Coefficient i.e. the service compliance for all services in the architectural component.

<u>Architectural Component Service Compliance Coefficient</u>:

The Architectural Component Service Compliance Coefficient, *ArchSvCoeff(d)*, is a measure of an asset component's compliance to the services (both provided and required) of a particular architectural component. It is the average of the Service Compliance Coefficient of all the services associated with a particular architectural component.

$$ArchSvCoeff(d) = \frac{\sum_{s\in DRACSv(d)} SvCoeff(s)}{|ArchSv(d)|} \quad (8)$$

Where,

ArchSv(d): Set of services (provided and required) for architectural component *d*.

Off course we can calculate separate coefficients for *provided* and *required* service by setting ArchSv(d) to the set of provided services or required services only.

4.1.2 Architectural Component Attribute Compliance Coefficient

The Architectural Component Attribute Compliance Coefficient, ArchAttrCoeff(d), is a measure of an asset component's compliance to all the data attributes of a particular architecture component. It is essentially the average of the components compliance to each of the attributes that it is registered to. ArchAttrCoeff(d) is measured in terms of the Data Attribute Compliance Coefficient or AttrCoeff(a). AttrCoeff(a) measures the extent to which an asset component is compliant with component data as specified in the architecture description. For each Data Attribute *a* in a architecture component, AttrCoeff(a) is calculated as:

$$AttrCoeff(a) = \frac{|Attr_{regd}(a)|}{|Attr(a)|} \quad (9)$$

Where

$Attr_{regd}(a)$: Set of elements in attribute *a* to which the component is registered.

$Attr(a)$: Set of all the elements of Attribute *a*.

Finally we calculate, the ArchAttrCoeff(d) which is the average of the Data Attribute Compliance Coefficient of all the attributes associated with a particular architectural component.

$$ArchAttrCoeff(d) = \frac{\sum_{a\in ArchAttr(d)} AttrCoeff(a)}{|ArchAttr(d)|} \quad (10)$$

Where

ArchAttr(d): Set of Attributes in architecture component d

4.1.3 Architectural Component Behaviour Compliance Coefficient

The Architectural Component Behavior Compliance Coefficient measures the degree of compliance of an asset component to the behavior of an architectural component captured in the architecture descriptions. It is measured in terms of the Behavioral Unit Coefficient *BehavUnitCoeff(bu)*, where *BehavUnitCoeff(bu)* is computed as below

$$BehavUnitCoeff(bu) = \frac{|BehavUnitEl_{regd}(bu)|}{|BehavUnitEl(bu)|} \quad (11)$$

Where

$BehavUnitEl_{regd}$(bu): Set of behavioral unit elements the component is registered to.

BehavUnitEl(bu): Set of elements in a particular behavioral unit, where an element is one of the quintuples – State, Trigger, Guard, Effects and Target.

With the above, we calculate the Architecture Component Behavior Compliance Coefficient

$$ArchBehavCoeff(d) = \frac{\sum_{bu\in ArchBehavUnit(d)} BehavUnitCoeff(bu)}{|ArchBehavUnit(d)|} \quad (12)$$

ArchBehavUnit(d): Set of Behavioral Units of architectural component *d*.

4.1.4 Architectural Component Compliance Coefficient

Now using the Service Compliance Metric evaluated for each service, the Data Attribute Compliance

Coefficient is evaluated for each attribute and the Behavioral Compliance Coefficient, the notion of a component's overall compliance to a architectural component can be calculated. The Architectural Component Compliance Coefficient, *ArchCoeff(d)*, measures the asset component's overall compliance to an architectural component.

$$ArchCoeff\ (d) = \frac{|ArchSv\,(d)| \times ArchSvCoeff\,(d) + |ArchAttr\,(d)| \times ArchAttrCoeff\,(d) + |ArchBehavUnit(d)| \times ArchBehavCoeff\,(d)|}{|AcchSv\,(d)| + |ArchAttr\,(d)| + |ArchBehavUnit(d)|} \quad (13)$$

As mentioned previously, these coefficients measure an asset component's compliance to various elements of the architectural description. Therefore, if component A has a *SvCoeff* of 0.5 for a certain service in the domain and component B has a *SvCoeff* of 0.75 for the same service, it can be inferred that asset component B is more compliant to the architecture component and should be preferred over asset component A for that particular service implementation. On the same lines, if the goal is to implement a particular architectural component as a whole complying with the specified boundaries of functionality and data set forth by the Architectural Component, then a component with a higher *ArchCoeff (d)* should be preferred.

4.2 Component Characteristic Metrics

While the Architecture Compliance Metrics calculated the compliance of an asset component to the various elements of the architectural description individually, these metrics did not capture the characteristics of a particular component as a whole. The Component Characteristic metrics address this aspect. These metrics evaluate characteristics of an asset component with respect to component functionality, component data and the component as a whole. The Component Characteristic metrics measure the compliance and coverage of an asset component with respect to the component data (characterized by all the *attributes* in the architectural description) and component functionality (characterized by all the *services* in the architectural description).

The Component Characteristic Metrics are of three types (i) *The Proximity Metrics, (ii) The Compliance Metrics & (iii) The Coverage Metrics*. These metrics are discussed in the following sub-sections and they leverage the Architecture Compliance metrics derived in the previous section.

4.2.1 Proximity Metrics

The Proximity metrics are defined to measure "closeness" of two versions of an asset component with respect to a) component functionality i.e. *Interfaces/Services* b) component data i.e. *Attributes*. In essence, these coefficients indicate the proximity of two asset versions with respect to the architectural description. The utility of these metrics lies in the fact that they give an insight into how a component has evolved in terms of domain data and domain functional requirements.

Though the proximity metrics have been defined to measure "closeness" between two versions of the same asset component, the idea can be extended to measure proximity between two different components as well.

<u>Functional Proximity Metrics</u>

The Functional Proximity Metrics leverage the Functional Model (the collection of services contained in the architectural description) to measure the similarity between two components with regard to the functional requirements the components satisfy.

Let FC be the Functional Model Compliance Matrix representing the compliance of different versions of a component, *tc*, to the services of the architectural description. Thus the matrix FC for two versions of component *tc* can be represented as

$$\mathbf{FC=}\begin{array}{c} \\ v_1 \\ v_2 \end{array}\begin{array}{c} \quad s_1 \qquad\qquad s_2 \qquad\qquad\qquad s_n \\ \begin{bmatrix} \mathrm{SvCoeff}_{v_1}(s_1) & \mathrm{SvCoeff}_{v_1}(s_2) & \dots & \mathrm{SvCoeff}_{v_1}(s_n) \\ \mathrm{SvCoeff}_{v_2}(s_1) & \mathrm{SvCoeff}_{v_2}(s_2) & \dots & \mathrm{SvCoeff}_{v_2}(s_n) \end{bmatrix}\end{array} \quad (14)$$

Where

v_1 and v_2 represents the two versions of the component, *tc*.

$s_1 \dots s_n$ represents the list of services from a particular architecture description.

$SvCoeff_{v1}(s_n)$, $SvCoeff_{v2}(s_n)$ represents the Service Compliance Coefficient of version v_1 and v2 of the Component with the set of *Services*, s_n, in the architectural description.

Now, the Proximity Matrix with respect to the domain functionality, PMF, is defined as

$$\mathbf{PMF = [FC][FC]^T}$$

Where $[FC]^T$ denotes the transpose of the matrix FC.

The element $PMF_{i,j}$ represents the proximity of versions *i* and version *j* with respect to the Functional Model. The matrix PMF is not normalized. We use the Euclidean Vector Norm to normalize the matrix PMF. The normalized PMF restricts the value of the element $PMF_{i,j}$ between zero and one. The formalized notation for deriving a normalized PMF using the Euclidean Vector norm is given below.

The Functional Model Compliance Matrix for the software component, *tc*, can be written as:

FC = [s_{vt}] $_{v = 1 ....V,\ t = 1...T}$

Where s_{vt} represents the Service Compliance Coefficient for version, v, of the Component, *tc*, for service, s, in the architectural description. V represents the total number of versions of *tc* and T represents the total number of services in the architectural description.

The Service Compliance vector, $\mathbf{s_v}$, is represented as

$\mathbf{s_v} = [\mathbf{s_{v,1}}\ \ \mathbf{s_{v,2}}\ \ \mathbf{s_{v,3}}\ \ ....\mathbf{s_{v,T}}]$.

We know from the Euclidean Vector Norm that

$$\| \mathbf{s_v} \|_2 = \left(\sum_{t=1}^{F} s_{vt}^2 \right)^{1/2}$$

Using the Functional Model Compliance matrix, FC, it is now possible to define the proximity of two versions, say (*u*, *v*) of a component. The FC can be evaluated as the cosine of the angle formed by vectors $\mathbf{s_v}$ and $\mathbf{s_u}$ that can be computed as the dot product of $\frac{s_v}{\| s_v \|_2}$ and $\frac{s_u}{\| s_u \|_2}$, respectively. Thus the Normalized Proximity Matrix, PMFN, can be represented as

PMFN = [f_{uv}] $_{u = 1 ....V,\ v = 1...V}$ (15)

where $\mathbf{f_{uv}} = \sum_{t=1}^{F} \left(\frac{s_{vt}}{\| s_v \|_2} \right) \left(\frac{s_{ut}}{\| s_u \|_2} \right)$

After normalization, we are assured that $0 \leq f_{uv} \leq 1$. At the boundaries, the following interpretations can be made:

$\mathbf{f_{uv} = 1}$: The versions *u* and *v* are exactly similar in terms of the Functional Model.

$\mathbf{f_{uv} = 0}$: There is no similarity in the functional model coverage of the two versions of the component. In other words they satisfy non-overlapping sets of services in the architectural description.

The higher the value of $\mathbf{f_{uv}}$ the more similar the two versions of the component are and lower the value, the more dissimilar the two versions are with respect to functional requirements.

Data Proximity Metrics

The Data Proximity Metrics leverage the Data Model (the collection of data attributes contained within the architectural description) to measure the similarity between two components with regard to the data requirements the components satisfy. The treatment used for deriving the proximity metrics with respect to the Data Model is similar to the one used for deriving the proximity metrics with respect to the Functional Model. The only difference lies in the fact that the Data Model Compliance matrix, DC, is represented by

$$\mathbf{DC} = \begin{array}{c} \\ v_1 \\ v_2 \end{array} \begin{array}{cccc} a_1 & a_2 & & a_n \\ \left[\begin{array}{c} \mathrm{AttrCoeff}_{v_1}(a_1) \\ \mathrm{AttrCoeff}_{v_2}(a_1) \end{array} \right. & \begin{array}{c} \mathrm{AttrCoeff}_{v_1}(a_2) \\ \mathrm{AttrCoeff}_{v_2}(a_2) \end{array} & \begin{array}{c} ... \\ ... \end{array} & \left. \begin{array}{c} \mathrm{AttrCoeff}_{v_1}(a_n) \\ \mathrm{AttrCoeff}_{v_2}(a_n) \end{array} \right] \end{array} \quad (16)$$

Where

v_1 and v_2 represent the versions of the Component, tc.

$a_1 \ldots a_n$ represents the list of attributes for a particular architectural description.

$AttrCoeff_{v1}(a_n)$ represents the Attribute Compliance Coefficient of version v1 of the component with the attribute a_n in the architectural description.

All other derivations and implications are exactly the same as the proximity metrics for the Functional Model.

4.2.2 Component Compliance Metrics

These metrics measure the compliance of an asset component to the architectural description as a whole. The Compliance Metrics are of two types – the Static Compliance metrics and the Compliance Evolution metrics. The Static Compliance metrics measure the degree of compliance of a component to the System Data Model and the System Functional Model. The Compliance Evolution metrics measure the percentage change of a component from one version to another in terms of system data and functionality.

Static Compliance Metrics

The Static Compliance metrics are termed 'static' as they measure the compliance of a given version of a component with respect to the Data and Functional Model.

The Data Model Compliance Index, DCmI(v), for a version of a component measures the compliance to the complete Data Model of the System. It is calculated as

$$\mathbf{DCmI(v)} = \frac{\sum_{a \in \mathrm{RegAttr(v)}} \mathrm{C(a)\ .\ AttrCoeff_v\ (a)}}{|\ \mathrm{RegAttr(v)}\ |}, \quad (17)$$

while the Functional Model Compliance Index

$$\mathbf{FCmI(v)} = \frac{\sum_{s \in \mathrm{RegSvc(v)}} \mathrm{C(s)\ .\ SvCoeff_v\ (s)}}{|\ \mathrm{RegSvc(v)}\ |} \quad (18)$$

Where

RegAttr(v)/ RegSvc(v): Set of Attributes/Services in the architectural description to which the version v of the component is registered.

C(a)/ C(s): Criticality of the Attribute/Service a/s in the domain.

AttrCoeff$_v$(a) SvCoeff$_v$(s): Attribute/Service Compliance Coefficient for Attribute/Service a/s for version v of the component.

The Criticality of the attribute/service is taken into account to reflect the relative importance in the System. If information regarding the criticality of data or services does not exist or is not specified, C(s) and C(a) should be specified as 1.

The System Model Compliance Index, which is the measure of a component's overall compliance to the domain requirements as represented in the architectural description. It is calculated as

$$\textbf{SysCmI(v)} = \frac{DCmI(v) + FCmI(v)}{2} \quad (19)$$

In a situation where a system integrator has two components to evaluate for satisfying a given functionality, he/she should select a component with the higher value of SysCmI (v) if overall domain compliance is desired.

<u>Compliance Evolution Metrics</u>

The Compliance Evolution metrics for a component are intended to measure the percentage change in the component's compliance to the domain from one version to another.

The Data Model Compliance Evolution Index, **DCmE (v_{new}, v_{base})**, for a component, captures the percentage change in the compliance to the Data Model from one version to another and is calculated as

$$\textbf{DCmE}(v_{new}, v_{base}) = \frac{DCmI(v_{new}) - DCmI(v_{base})}{DCmI(v_{base})} \times 100 \quad (20)$$

Similarly, the Functional Model Compliance Evolution Index

$$\textbf{FCmE}(v_{new}, v_{base}) = \frac{FCmI(v_{new}) - FCmI(v_{base})}{FCmI(v_{base})} \times 100 \quad .(21)$$

The rollup of data and functional evolution, the System Model Compliance Evolution Index, SysCmE(v_{new}, v_{base}), for a component, captures the percentage change in the component's compliance to the overall domain requirements from one version to another and is calculated as in below.

$$\textbf{SysCmE}(v_{new}, v_{base}) = \frac{SysCmI(v_{new}) - SysCmI(v_{base})}{SysCmI(v_{base})} X100 \quad (22)$$

In a typical software system, we are likely to see high positive values for compliance evolution indices, which would indicate that the new version of the component is more compliant to the domain. A negative value would indicate that the new component has lower compliance to the domain, which in most general cases is not desirable.

4.2.3 Component Coverage Metrics

The Component Coverage Metrics measure how much of the Data and the Functional Model is covered by a component. Similar to the Compliance metrics, the Coverage metrics are of 2 types (i) The Static Coverage Metrics and (ii) The Coverage Evolution Metrics. These metrics are explained below.

<u>Static Coverage Metrics</u>

The Static Coverage Metrics measure the coverage of the Functional & Data Model by a *particular* version of a software component. These are very simple indices which capture how much of a domain is covered by a component and is intended to facilitate decision making with respect to the scope of individual components for a system deployment.

The Data Model Coverage Index,

$$\textbf{DCvI(v)} = \frac{|RegAttr(v)|}{|RAAttr(ra)|} \quad (23)$$

while the Functional Model Coverage Index,

$$\textbf{FCvI(v)} = \frac{|RegSvc(v)|}{|RASvc(ra)|} \quad (24)$$

Where

RegAttr(v)/ RegSvc(v): Set of attributes/services in the architectural description to which the version v of the component is registered.

RAAttr(ra)/ RASvc(ra): Set of all the attributes in the architectural description, ra.

The overall System model Coverage Index with respect to both functions and data is calculated as

$$\textbf{SysCvI(v)} = \frac{DCvI(v) + FCvI(v)}{2} \quad (25)$$

The Static Coverage metrics support the first level decision making for short listing of potential asset components for a given set of functionality. After the shortlist is made, the component compliance metrics, explained in the previous section, should be used for identifying the most suitable candidate.

<u>Coverage Evolution Metrics</u>

The Coverage Evolution Metrics measures the percentage change in the coverage of the Functional & Data Model as well as the overall System Model as a component evolves from one version to another. These metrics are intended to facilitate system evolution decisions.

The Data Model Coverage Evolution Index measures the change in the coverage of the Data Model as the component evolves from version $\mathbf{v_{base}}$ to $\mathbf{v_{new}}$ and is calculated as:

$$\mathbf{DCvE}\ (\mathbf{v_{new}}, \mathbf{v_{base}}) = \frac{DCvI(v_{new}) - DCvI(v_{base})}{DCvI(v_{base})} \times 100 \quad (26)$$

The Functional Model Coverage Evolution Index, which measures the change in coverage of the Functional Model from version $\mathbf{v_{base}}$ to $\mathbf{v_{new}}$ is computed as:

$$\mathbf{FCvE}\ (\mathbf{v_{new}}, \mathbf{v_{base}}) = \frac{FCvI(v_{new}) - FCvI(v_{base})}{FCvI(v_{base})} \times 100 \quad (27)$$

Similar to the Static Compliance Metrics, the overall System Model Coverage Evolution Index, which measures the percentage change in the overall coverage of the domain requirements between two versions of the component is calculated as

$$\mathbf{SysCvE}(\mathbf{v_{new}}, \mathbf{v_{base}}) = \frac{SysCvI(v_{new}) - SysCvI(v_{base})}{SysCvI(v_{base})} X100 \quad (28)$$

The Coverage Evolution indices help track the functional change in a component from one version to the next. Off course these metrics are equally applicable for comparing different applications, when used in the correct context.

5. Application of the Metrics

Reusable components should ideally have high values for the Compliance and Coverage metrics. Target threshold values, which may come from Program Managers or System Integrators, could be used as design drivers when a reusable component is being built from scratch. The goal in such a case should be to aim for

(i) High compliance to architecture functionality,
(ii) High compliance to architecture data, and
(iii) High compliance to architecture component description.

For asset components, the Architecture Compliance metrics provide a useful mechanism for evaluating reuse potential of these components and help in decision-making about suitability of reuse candidates.

These metrics were applied in a sample University Registration System where the architectural description consisted of 45 services and 22 data entities distributed over 15 architectural components. The returned results were quite satisfactory. Detailed elaboration of the experiment cannot be provided in this paper due to constraints of space. We present a single result for demonstrating the concept.

Service Compliance Coefficient for two reusable components TEX1.0 and ROSEv1.0 were calculated. The Service Compliance Coefficient (*SvCoeff*) of TEX v1.0 was 0.84 while that of ROSE v1.0 was 0.63 (Table 1). The values are calculated using formula (7). Thus for the given service "Add a Class", TEX 1.0 was selected over ROSE v1.0 as it satisfied the functional requirement better.

Table 1: Calculation of Service Compliance

System Architecture: *Student Registration*		
Arch. Component Name: *Registration System*		
Service Name: *Add a class*		
	TEX v1.0	ROSE v1.0
Input Data Compliance Coefficient – IDCoeff(s)	0.94	0.89
Input Data Dependency – IDDep(s)	1	1
Output Data Compliance Coefficient – ODCoeff(s)	0.75	0.45
Output Data Dependency – ODDep(s)	1	1
Input Event Compliance Coefficient - IECoeff(s)	1	0.5
Input Event Dependency – IEDep(s)	1	1
Output Event Compliance Coefficient - IECoeff(s)	0.67	0.67
Output Event Dependency – IEDep(s)	1	1
Service Compliance Metric – SvCoeff(s)	0.84	0.63

By considering the Service Compliance Coefficient of similar components, component developers would be able to identify whether their component is competitive enough (i.e. competitive against other components for the domain) for a particular service implementation and therefore take necessary steps to increase the reuse potential of their components.

It's worth mentioning in this context that we are currently in the process of building a tool that would automate the registration process and calculate the metrics for architectural evaluation.

6. Conclusion

The contextual metrics presented here provide a mechanism for a quantitative evaluation of software component reuse in the context of architecture requirements (functional and data) and architecture structure. We leverage the requirements represented within an architectural description to provide the context for an asset component to evaluate the compliance of these components to the architectural description, and to assess the similarity between components, the component's coverage of the architectural description, as well as numerically tracking the evolution of a component in terms of the architectural description. Not only do our metrics provide objective measures in the context of the architecture, they also enable quantitative decision-making regarding the behavior of components. Our reusability assessment thus goes beyond simple interface matching and helps system integrators explore behavioral characteristics of components as well.

These metrics are '*generic metrics'* as the measurement indices are not constrained by the nature of the components being evaluated and can be applied to any component. Defining "generic metrics" has been one of the recognized goals of the reuse research community. These metrics provide simple yet realistic, quantitative measure of reuse potential from a domain perspective and help evaluate the benefits of selecting one component over another at design time. They are intuitive in nature, are consistent, reproducible, and can be used to provide meaningful insight for various system stakeholders.

7. References

[1] Perry, D. E., Wolf, A. L., "Foundations for the Study of Software Architectures", ACM Software Engineering Notes, 17, 4, October 1992, 40-52

[2] Prieto-Diaz, Ruben, and Peter Freeman, "Classifying Software for Reusability," IEEE Software, Vol. 4, No. 1, January 1987, pp. 6-16.

[3] Selby, Richard W., "Quantitative Studies of Software Reuse," in Software Reusability, Volume 2, Ted J. Biggerstaff and Alan J. Perlis, eds. Addison-Wesley, Reading, MA, 1989.

[4] Karlson, Even-Andre, Guttorm Sindre, and Tor Stalhane, "Techniques for Making More Reusble Components,"REBOOT Technical Report #41, 7 June 1992

[5] Jeffrey S. Poulin, "Measuring Software Reuse – Principles, Practices and Economic Models", Addison-Wesley, 1996

[6] Caldiera, Gianluigi, and Victor R. Basili, "Identifying and Qualifying Reusable Software Components," IEEE Computer, Vol. 24, No. 2, February 1991, pp. 61-70.

[7] Hooper, James W., and Chester, Rowena O., Software Reuse Guilines and Methods. Plenum Press, NY, 1991.

[8] Boetticher, G.,K. Srinivas, and D. Eichmann "A Neural Net-based Approach to Software Metrics," Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering, San Francisco, CA, 14-18 June 1993, pp. 271-274,

[9] Tracz, Will, "Software Reuse Maxims," ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 4, October 1998, pp. 28-31

[10] Torres, William R., and Mansur H. Samadzadeh, "Software Reuse ad Information Theory Based Metrics," Proc. 1991 Symposium on Applied Computing (SAC '91), Kansas City, MO, 3-5 April 1991, pp.437-446.

[11] Edwards, Stephan, "An Approach for Constructing Reusable Software Components in Ada," Strategic Defense Organization Pub #Ada233 662, Washington, D. C., September 1990.

[12] Hooper, James W, and Chester, Rowena O., Software Reuse Guidelines and Methods. Plenum Press, NY, 1991

[13] Hollingsworth, Joe, Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada. Ph. D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1992.

[14] NATO, "Standard for the Development of Reusable Software Components," NATO Communications and Information Systems Agency, 18 August 1991.

[15] Sommerville, I., L. Masera, and C. Dmaria, "Practical Guidelines for Ada Reuse in an Industrial Envronment, "Proceedings of the Second Symposium on Software Quality Techniques a Acquisition Criteria, Florence, Italy, 29-31, May 1995, pp. 138-147. URL: htp://www.comp.lancs.ac.uk/computing/research/cseg/projects/APPRAISAL/

[16] Khairuddin, H., and Elizabeth K., "A Software Reusability Attributes Model," Journal of Computer Aided Technology, Vol. 8, No. 1-2, 1995, pp. 69-77

[17] Tracz, Will, "A Conceptual Model for Mega programming," ACM SIGSOFT Software Engineering Notes, Vol. 16, No. 3, July 1991, pp. 36-45.

[18] Paul R. A., Metrics guided Reuse, Proceedings of the Seventh International Conference on Tools with Artificial Intelligence,1995.

[19] K. S. Barber and Sutirtha Bhattacharya, "Representing Technology to Promote Reuse in the Software Design Process," In Proceedings of the 15th IEEE Conference on Automated Software Engineering, 2000, Grenoble, France, 11-15 September 2000, pages 285-288.

[20] Barber, K. S. and Bhattacharya, S."A Representational Framework for Technology Component Reuse," 13th International Conference on Software & Systems Engineering and their Applications (ICSSEA'2000), Volume 1, Session 2, Section 1, December 5–8, 2000, Paris, France.

[21] Perry, D. E.,"Generic Descriptions for Product Line Architectures". *ARES II Product Line Architecture Workshop*, Los Palmos, Gran Canaria, Spain, February 1998

[22] Habermann, A. N., Perry, D. E., "Well Formed System Composition. Carnegie-Mellon University, Technical Report CMU-CS-80-117. March 1980

[23] Perry, D. E., ``The Inscape Environment: A Practical Approach to Specifications in Large-Scale Software Development. A Position Paper." January 1990.

[24] D. C. Luckham, D. C., Vera, J., "An Event-Based Architecture Definition Language." IEEE Transactions on Software Engineering, vol. 21, no. 9, pages 717-734, September 1995.

[25] Bhattacharya, S. "Specification and Evaluation of Technology Components to Enhance Reuse," Masters Thesis, The University of Texas at Austin, July 2000