Architecture Assessment Model for System Evolution

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

Abstract[1]

Even though there has been some research on system evolution, there is no well defined vocabulary to indicate deviation of a system from desired goals. Further, there are no objective measures to indicate whether changes incorporated into a system as part of its evolution violates the integrity of the architectural design. Also, little research has been done to categorize the aspects of a software system that is subject to deviation as the system evolves.

In this paper we develop a model for tracking software evolution and propose measures that will objectively indicate the extent of deviation or divergence in a software system. We also categorize the different aspects of software, changes to which can significantly impact usability as well as conceptual coherence.

1. Introduction

Most software systems undergo significant changes over their lifetime. In fact it is common experience that any software without an active roadmap soon falls out of favor with its users. Several reasons can be identified for such change

- System requirements evolve
- The context in which the software operate changes
- New capabilities that build on existing ones are identified
- Maintenance activities are performed keep the software operational

Ideally, all system change should be reified in the architecture first followed by implementation changes or enhancements. However, it has commonly been observed that due to time-to-market pressures as well as sub-optimal development processes, the changes to a system often *erode* the fundamental characteristics of the original architecture which progressively results in intractable systems that fail to satisfy the basic reliability, availability and performance requirements.

Although the concept of architectural erosion was identified a decade and a half ago [1], approaches for assessment of loss of architectural characteristics have focused on *architectural recovery*, where architectural information is recovered from source code [e.g. 2, 3, 4, 5, 6]. However, since architecturally relevant information can be obscured in the implementation and implementations often violate system requirements, Medvidovic et al. has proposed a lightweight approach for correlating implementations with software requirements using architectural styles [7]. We believe that while checking on deviations using the actual implementation is necessary, relying solely on the implementation for this assessment pushes an important aspect of system evolution to a very late phase in the development process, when corrective actions could be significantly more expensive. While O'Rielly et al. [8] proposed a model based approach, there was still a reliance on implementation and no metrics were proposed.

In this paper we propose a model for assessing architectural deviation based on architectural model changes and propose measures that will objectively indicate the extent of change in a software system. Our model will also highlight loss of system functionality and architectural structure. As part of the model development, we categorize the different aspects of software, changes to which can significantly impact usability and conceptual coherence.

In the second and third sections of this paper we discuss the background and the high level assessment model. In the next two sections we develop our proposed measures for system evolution. The last two sections discuss the validation of our proposal and the conclusions.

2. Background

The proposed assessment model for system evolution is based on our abstract architectural model representation [9]. We discuss the representation briefly.

The abstract architectural model partitions the architectural representation into Architectural Functionality Spec and Architectural Non-functional Spec. We will

[1] This research is supported in part by NSF CISE grant IIS-0438967. Please note that any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

elaborate on the Architectural Functionality Spec as the system evolution assessment is based on it.

The Architectural Functionality Spec is composed of Architectural Component Specs. These component specs capture the specification of the key components of the architecture. The intent of these components is to partition the system domain into logical abstractions. Each Architectural Component Spec is qualified by the Interface Spec, the Attribute Spec and the Behavioral Spec. The Interface Spec is composed of the Provided Services Spec and the Required Services Spec and captures the service interfaces the component exposes as well as the ones it needs. The Attribute Specs capture the data managed by the component while the Behavioral Specs capture the behavioral model using a state chart representation. The Interface Specs (Provided and Required) comprehend the input and output data, the input and output events as well as the pre and post conditions associated with each service interface.

The *form* of the architecture is captured in our abstract model in terms of Form Units where an individual Form Unit is represented by one or many Form Unit Mapping(s). Each Form Unit Mapping is qualified by an Architectural Component pair, the cardinality associated with the Architectural Elements and the Rule that explains the relationship between the Architectural Element pair.

3. High Level Assessment Model

All systems, independent of the application domain, are built to deliver a set of services for end users. Fundamental to the delivery of these services is the architectural integrity with which these services are delivered. Experience with decades of software development has proved that software architecture has more to do with utility than aesthetics – much more so than its civil engineering counterpart. A poorly constructed system may be able to satisfy immediate functional requirements, but it may fail to satisfy reliability, availability or performance requirements. Even if a system can satisfy the desired non-functional requirements, it may be difficult to modify and enhance the system. It is therefore extremely critical that software is effectively architected to satisfy both functional and non-functional goals. Also, since typical software systems go through regular enhancements and releases, an architecture not designed for extensibility and evolution will fail to deliver desired capabilities in the long run. Given the inevitability of evolution of a software system, it is important that an assessment model for system evolution is developed at an architectural level.

While non-functional requirements are a critical part of any software architecture, the *raison d'etre* for any software system is delivery of functional requirements. However the structure of the system influences a number of the non-functional requirements. This research hypothesizes that focusing on the functional and structural implications of software architecture evolution are adequate to assess desirable and undesirable deviations from key architectural goals. Of course, deviation needs to be measured both in terms of change from an architectural blueprint as well as from one software version to another. This is because in many software development projects there are no documented architectural blueprints, and we do need a baseline for the assessment of deviation. Fundamental to our approach is the concept of *registration.* Registration is defined as the process of formally associating component characteristics to the architectural blueprint and establishing mappings in a form that captures the association and is computable.

We define deviation from an architectural baseline in terms of divergence. Measurement of divergence is done at the level of individual services and data attributes. We elaborate on the assessment of divergence in section 4. The fundamental idea behind the measurement of divergence is the change in the *registration* of various architectural elements to the base architecture.

Most complex software systems undergo several rounds of developer turnover and in most industry projects it is seen that the initial designers and developers are no longer in the team when critical enhancements need to be made. It is very important during such enhancements or modifications that useful functionality as well as the structural integrity of the system is not lost while delivering new capabilities. To identify such undesired change we introduce the notion of architectural erosion. Just as the washing away of the topsoil due to running rainwater results in soil erosion which leaves the land barren, architectural erosion results in the wearing away of useful capabilities from a system. The erosion indicators are of two types – functional and structural. These are elaborated further in Section 5.

It is to be noted that in the subsequent sections we make references to 'software versions' and 'implementations' during our elaboration of the divergence and erosion measures. This is because our approach is applicable for measuring deviation from either an architectural blueprint or from a previous version of a software component that has been specified against an architectural blueprint. While a development organization can use our approach to assess deviation post system construction, greater value would be derived if the assessment was done prior to actual construction, using the architectural specs for the component being built.

4. Architectural Divergence

One of the key concepts used in the assessment of system evolution is architectural divergence. Divergence essentially measures the change in compliance of an asset component with respect to an architectural blueprint. As a system evolves, it undergoes changes in the set of services it offers as well as the data it manages. In an ideal scenario new capabilities should be added to a component without loss of existing functionality, except in cases where certain services are intentionally deprecated. However, in real life systems, we often see that changes made in one portion of the software, affects some other portions. Though some of these are done intentionally, most of them are unintentional. A few examples are given below:

- An existing service is extended to deliver additional functionality and the API is modified, which breaks existing method invocation code
- Data Entities are extended with additional data elements which may break the consistency with an existing schema
- For event based systems, new events are introduced, or existing ones modified, which modifies system response

In the absence of a mechanism to assess such changes, bugs will be identified only during system testing. Fixing issues that late in the development process is typically quite expensive. It would be ideal to understand the impact of changes planned as early in the development cycle as possible. The benefit is maximum if it can be done at the architectural level. In this section we explain our proposal for the assessment of divergence at an architectural level.

Divergence is measured relative to a baseline. If the intent is to measure the divergence from the architectural blueprint, then the architectural blueprint needs to be considered as the baseline. For measuring the divergence between two different versions of an asset component, the earlier version should be treated as the baseline.

Divergence can be measured both from a functional perspective as well as from a structural perspective. Functional divergence measures how a given version of software deviates from the baseline in terms of the architectural services and data supported. Structural divergence on the contrary measures the deviation in terms of architectural characteristics captured in the *Form* of the architectural description of our abstract architecture model.

We describe Functional and Structural Divergence in the next two subsections.

4.1 Functional Divergence

Functional Divergence measures the divergence with respect to services and the data captured in the architectural description. To measure service divergences, divergence for the associated input and output data, the input and output events and the pre and post condition need to be measured. The service level divergence can be bundled to quantify the overall divergence from all the services associated with an architectural component. We also measure divergence in terms of data attributes managed by an architectural component. Individual attribute divergence can be bundled for an assessment of the overall architectural component attribute divergence. A similar treatment can be applied to measure behavioral divergence. The bundled service divergence, attribute divergence and behavioral divergence can be used to compute an overall architectural component divergence.

The Functional Divergence measures are explained below

Service Level Divergence

In our abstract architectural model, a service is modeled by the input data that are required for the execution of the service, the output data that are generated by the service, the input events that trigger the service, the output events that are generated by the service, the pre-condition that need to be satisfied before the service executes and post conditions that need to be reflected in the component state to indicate satisfactory execution of the service. Therefore, to measure the service level divergence all of the above aspects need to be considered. We explain each of these below.

Input and Output Event Divergence:

Input and Output Event Divergences are measured for a given service. Any reference to divergence for events will correspond to a particular service, s, from the architectural description. We explain the derivation of Input Event Divergence IEDiv(s) in details. The derivation of Output Event Divergence, OEDiv(s), is analogous. The key components that are used for the measurement of IEDiv(s) are the two sets $\text{RegdIE}_{new}(s)$ and $\text{RegdIE}_{base}(s)$. $\text{RegdIE}_{new}(s)$ capture the *registration* information about the input events associated with the service s for the asset component that is classified as *new*. Similarly the set, $\text{RegdIE}_{base}(s)$ captures the registration information for the component classified as *base*. If the service deign or implementation (in case assessment is done after system construction) for the component accepts the particular event as an input as specified in the architectural description, the corresponding element of the set is set to one, if it does not, the element is set to zero. So essentially the sets $\text{RegdIE}_{new}(s)$ and $\text{RegdIE}_{base}(s)$ are a collection of ones and zeros, with each element signifying the registration or otherwise for each event associated with the service. It is obvious that the cardinality of both of these sets are the same as the set of events (input events in this case) associated with the service are defined in the architectural specification which is independent of any specific component implementation. We represent the registration information for a particular event, ev, for the service, s, by RegdIE(s, ev). It corresponds to either a one (when the designed/implemented service is triggered by event ev in the architecture) or zero (the designed/implemented service is not triggered by the event ev). Therefore the factor $[\text{RegdIE}_{new}(s, ev) - \text{RegdIE}_{base}(s, ev)]$ corresponds to a zero if the service design/implementation for both the *new* and the *base* component are the same wrt triggering by event ev, or one, if there is a change from one component to another. Since we are interested in the overall divergence, we consider the absolute value i.e. $\text{Abs}[\text{RegdIE}_{new}(s, ev) - \text{RegdIE}_{base}(s, ev)]$. The deviation for each event associated with the service s is summed over all the events (represented by the set IE(s)) and the resultant sum is divided by the cardinality of the set IE(s), which corresponds to the number of input events associated with the given service in the architectural description.

IEDiv(s) and OEDiv(s) are shown below:

$$IEDiv(s) = \frac{1}{|IE(s)|} \sum_{ev \in IE(s)} \begin{matrix} Abs[RegdIE_{new}(s, ev) \\ - RegdIE_{base}(s, ev)] \end{matrix} \quad (1)$$

$$OEDiv(s) = \frac{1}{|OE(s)|} \sum_{ev \in OE(s)} \begin{matrix} Abs[RegdOE_{new}(s, ev) \\ - RegdOE_{base}(s, ev)] \end{matrix} \quad (2)$$

Where

IE(s)/ OE(s): Set of I/O events for service s

$RegdIE_{new}$(s)/ $RegdOE_{new}$(s): Registration information for input/output events corresponding to service s for *new*

$RegdIE_{new}$(s, ev)/ $RegdOE_{new}$(s, ev): Value of element for event, ev, in the set $RegdIE_{new}$(s)/ $RegdOE_{new}$(s)

$RegdIE_{base}$(s)/ $RegdOE_{base}$(s): Registration information for I/O events corresponding to service s for *base*

$RegdIE_{base}$(s, ev)/ $RegdOE_{base}$(s, ev): Value of element for event, ev, in the set $RegdIE_{new}$(s)/ $RegdOE_{new}$(s)

From above, $0 \leq$ IEDiv(s), OEDiv(s) ≤ 1. As an example, a value of 0.5 for IEDiv(s) would indicate that half the input events for a given service s has changed from the *base* version to *new*.

Input and Output Data Divergence:

The approach for the assessment of data divergence is similar to that of input and output events with some modifications to comprehend the way input and output data are represented. Input and output data are modeled by data entities which can be made up of several data elements. For example, the data entity 'Address' could be made up of several data elements such as 'Street Address', 'City', 'Zip', 'Country' etc. The approach is similar to the one used for database designs where tables represent the data entities in the schema while the fields represent the data elements.

We explain the derivation for Input Data Divergence, IDDiv(s), in details. The two sets $RegdIDEl_{new}$(s, en) and $RegdIDEl_{base}$(s, en) correspond to the registration information for the two components *new* and *base* with respect to the data entity, en, which is an input data entity for the service, s. The cardinality of these two sets equal the number of elements associated with the given data entity, en. We use $RegdIDEl_{new}$(s, en, el) and $RegdIDEl_{base}$(s, en, el) as the notation for representing the registration of the components *new* or *base* with the element el, in entity en, that is needed for execution of the service, s. The value is either a one (when the designed/implemented service requires the data element el in entity en for execution of service s) or zero (the designed/implemented service does not need the data element el in the entity en for execution). The factor [$RegdIDEl_{new}$(s, en, el) - $RegdIDEl_{base}$(s, en, el)] equals zero if the service design for both the *new* and the *base* components are the same wrt the need for the data element el in the data entity en, for execution of service s. It equals one, if there is a change from *base* to *new*. Since we are interested in the overall divergence, we consider the absolute value Abs[$RegdIDEl_{new}$(s, en, el) - $RegdIDEl_{base}$(s, en, el)]. This absolute value is summed up over all the elements associated with the data entity en. To calculate the divergence from *base* to *new* for a given entity, en, the sum of the absolute values for each data element is divided by the number of elements associated with the entity en. IDEl(s, en) is the set of elements associated with input data entity, en, and hence the cardinality of this set represents the number of elements associated with the entity en. The overall divergence for all the input data entities, IDDiv(s), is computed by taking the sum of the divergences for each data entity and dividing the sum by the number of input data entities for the service s as defined in the architecture.

The derivation of Output Data Divergence, ODDiv(s), is analogous. IDDiv(s) and ODDiv(s) are shown below.

$$IDDiv(s) = \frac{1}{|IDEn(s)|} \sum_{en \in IDEn(s)} \frac{1}{|IDEl(s,en)|} \sum_{el \in IDEl(s,en)} Abs[RegdIDElnew(s,en,el) - RegdIDbase(s,en,el)] \quad (3)$$

$$ODDiv(s) = \frac{1}{|ODEn(s)|} \sum_{en \in ODEn(s)} \frac{1}{|ODEl(s,en)|} \sum_{el \in ODEl(s,en)} Abs[RegdODEl_{new}(s,en,el) - RegdOD_{base}(s,en,el)] \quad (4)$$

Where

IDEn(s)/ ODEn(s): The set of I/O data entities for service s

IDEl(s, en)/ ODEl(s, en): The set of elements associated with the I/O data entity en for service s

$RegdIDEl_{new}$(s, en)/ $RegdODEl_{new}$(s, en): Registration info for elements of I/O data entity en, for service s for *new*

$RegdIDEl_{new}$(s, en, el)/ $RegdODEl_{new}$(s, en, el): Value of registration for element el, in data entity en, for service s in the set $RegdIDEl_{new}$(s, en)/ $RegdODEl_{new}$(s, en)

$RegdIDEl_{base}$(s, en)/$RegdODEl_{base}$(s, en): Registration info for elements of I/O data entity en, for service s for *base*

$RegdIDEl_{base}$(s, en, el)/ $RegdODEl_{base}$(s, en, el): Value of registration for element el, in data entity en, for service s in the set $RegdIDEl_{base}$(s, en)/ $RegdODEl_{base}$(s, en)

It is to be noted that IDDiv(s) and ODDiv(s) follow the relation $0 \leq$ IDDiv(s), ODDiv(s) ≤ 1.

Pre-Condition and Post-Condition Divergence:

The derivation of Pre and Post Condition divergences are similar to that of Input and Output Events. We explain the approach for derivation of Pre-Condition Divergence. The two sets $RegdPreC_{new}(s)$ and $RegdPreC_{base}(s)$ capture the registration information of the components *new* and *base* for the pre-conditions for the execution of service s. Each element of these two sets is either a one or a zero depending on conformance to the pre-conditions associated with the service s. $RegdPreC_{new}(s, c)$ and $RegdPreC_{base}(s, c)$ correspond to the value, one or zero, representing whether the service design/implementation satisfies the pre-condition c or not. The factor $Abs[RegdPreC_{new}(s, c) – RegdPreC_{base}(s, c)]$ is zero if there is no change in conformance to pre-condition c from the component *base* to component *new*. If there is a change in the conformance, this factor equals one. To compute the Pre-Condition Divergence, PreCDiv(s), we take the sum of the factor for all the pre-conditions associated with the service s and divide the result by the count of the pre-conditions for the service, s. The count of the pre-conditions is given by the cardinality of the set PreC(s) which represents the set of pre-conditions associated with the service s.

Post-Condition divergence, PostCDiv(s), is computed using the same approach as Pre-Condition divergence. The formulae for PreCDiv(s) & PostCDiv(s) are:

$$PreCDiv(s) = \frac{1}{|PreC(s)|} \sum_{c \in PreC(s)} \begin{matrix} Abs[RegdPreCnew(s,c) \\ - RegdPreCbase(s,c)] \end{matrix} \quad (5)$$

$$PostCDiv(s) = \frac{1}{|PostC(s)|} \sum_{c \in PostC(s)} \begin{matrix} Abs[RegdPostCnew(s,c) \\ - RegdPostCbase(s,c)] \end{matrix} \quad (6)$$

Where

PreC(s)/ PostC(s): Set of pre/post-conditions for service s.

$RegdPreC_{new}(s)$/ $RegdPostC_{new}(s)$: Registration information for pre/post-conditions for service s for *new*

$RegdPreC_{new}(s, c)$/ $RegdPostC_{new}(s, c)$: Value of element for pre/post-condition c, for service s in the set $RegdPreC_{new}(s)$/ $RegdPostC_{new}(s)$

$RegdPreC_{base}(s)$/ $RegdPostC_{base}(s)$: Registration information for pre/post-conditions corresponding to service s for *base*

$RegdPreC_{base}(s, c)$/ $RegdPostC_{base}(s, c)$: Value of element for pre/post-condition c, for service s in the set $RegdPreC_{base}(s)$/ $RegdPostC_{base}(s)$

As in the case of event and data divergence, $0 \leq PreCDiv(s), PostCDiv(s) \leq 1$.

Service Divergence:

Service Divergence, SvDiv(s), for a service s measures the divergence wrt delivery of the service from component *base* to *new*. It takes into account divergences for input events, output events, input data, output data, pre-conditions and post-conditions. SvDiv(s) also comprehends the various dependencies for the generation and consumption of data, events and pre and post conditions. The various dependencies are used in this formula to highlight the relative importance of the respective deviations. A large deviation for the input data may not be a big issue if just one service generates all the required input data, however if many services are involved in the generation of the input data the potential impact is large, as much more scrutiny will be needed to account for the deviation. SvDiv(S) is really the weighted average of the Input Data Divergence IDDiv(s), the Output Data Divergence ODDiv(s), the Input Event Divergence IEDiv(s), the Output Event Divergence OEDiv(s), the Pre-Condition Divergence PreCDiv(s) and the Post-Condition Divergence PostCDiv(s) where the respective weights are the various dependencies. The value of SvDiv(s) is bounded by $0 \leq SvDiv(s) \leq 1$.

The formula for Service Divergence SvDiv(s) is:

$$SvDiv(s) = \frac{\begin{matrix} IEDep(s)xIEDiv(s) + OEDep(s)xOEDiv(s) + \\ IDDep(s)xIDDiv(s) + ODDep(s)xODDiv(s) + \\ PreCDep(s)xPreCDiv(s) + PostCDep(s)xPostCDiv(s) \end{matrix}}{\begin{matrix} IDDep(s) + ODDep(s) + IEDep(s) + OEDep(s) + \\ PreCDep(s) + PostCDep(s) \end{matrix}} \quad (7)$$

Where

IEDep(s): Input Event Dependency i.e. count of scenarios that generate the events that trigger service s

IEDiv(s): Input Event Divergence for service s

OEDep(s): Output Event Dependency i.e. count of services that consume events generated by service s

OEDiv(s): Output Event Divergence for service s

IDDep(s): Input Data Dependency i.e. the count of services that generate the data required by service s

IDDiv(s): Input Data Divergence for service s

ODDep(s): Output Data Dependency i.e. count of services that consume the data generated by service s

ODDiv(s): Output Data Divergence for service s

PreCDep(s): Pre-Condition Dependency i.e. count of services that establish pre-conditions for execution of service s

PreCDiv(s): Pre-Condition Divergence for service s

PostCDep(s): Post-Condition Dependency i.e. count of services that depend on post-conditions established by s.

PostCDiv(s): Post-Condition Divergence for service s

Architectural Component Service Divergence:

We combine the divergence evaluated for each service to compute the overall service divergence for an architectural component. The Architectural Component Service Divergence, ArchCompSvDiv(d), is the average of the Service Divergence of all the associated services. This metric is useful for evaluating the deviation of one version of a component from another with respect to the realization of an architectural component. As in the case of SvDiv(s) for ArchCompSvDiv(d), $0 \leq ArchCompSvDiv(d) \leq 1$.

The formula for Architectural Service Divergence is:

$$ArchCompSvDiv(d) = \frac{1}{|ArchCompSv(d)|} \sum_{s \in ArchCompSv(d)} SvDiv(s) \qquad (8)$$

Where
SvDiv(s): Service Divergence for service s
ArchCompSv(d): Set of services for component d.

Attribute Level Divergence

Just as the delivery of a certain service may undergo changes from one version of software to another, the attributes managed by an architecture can also undergo change. It is important that we establish some objective criteria to assess the departure of an attribute from either a baseline architecture or from a previous version of the software. Generating and consuming data in the correct format is often the key to successful software integrations. This is especially true in the case of Dataflow architectural styles such as Batch Sequential and Pipes and Filters. It is also true for Data Centered styles such as Repository and Blackboard. As an example, let us assume that two software applications integrate in such a manner that the "Address" data attribute generated by application A is consumed by another application B. Now let us suppose that in the initial version of Application A, the data attribute "Address" had 5 data elements – "Street", "City", "County", "Country" and "Zip Code". Application B used the "County" information from the "Address" data attribute to determine the county based demographics. However in a later version, the developers of A incorrectly assumed that the "County" information in the "Address" is redundant and dropped it. As a result B would fail to deliver the county based demographic information. This situation could be avoided if there was an objective way to determine that the later version of application A had *diverged* from the initial version with respect to support for the data attributes. The Attribute Level Divergence measures have been defined to address this gap.

Attribute Divergence:

Attribute Divergence, AttrDiv(a), measures the deviation with respect to the delivery of a given attribute a. For the "Address" example mentioned above, a review of AttrDiv(a) for the attribute would have identified the difference from one version to another, which could have subsequently been addressed without going through an integration failure.

For computing the Attribute Divergence, we use the two sets RegdAttr$_{new}$(a) and RegdAttr$_{base}$(a) which capture the registration information for the attribute a in the *new* version and the *base* version respectively. These sets are a collection of ones and zeros, with a one indicating that the application is registered to the corresponding element associated with the attribute, while a zero indicates that the attribute doesn't support the corresponding element. For each of the sets there will be as many entries as there are elements associated with the given attribute as specified in the abstract architectural model and therefore the cardinality of the two sets RegdAttr$_{new}$(a) and RegdAttr$_{base}$(a) will be the same. The absolute value of [RegdAttr$_{new}$(a, el) - RegdAttr$_{base}$(a, el)] is one if there is a change in the component with respect to the support of data element el, associated with the attribute a. The change or otherwise of each element is summed up over all elements of the attribute and divided by the total number of elements associated with the attribute to calculate AttrDiv(a). The formula for AttrDiv(a) is shown below

$$AttrDiv(a) = \frac{1}{|Attr(a)|} \sum_{el \in Attr(a)} Abs[RegdAttr_{new}(a,el) - RegdAttr_{base}(a,el)] \qquad (9)$$

Where
Attr(a): Set of data elements associated with attribute a
RegdAttr$_{new}$(a): Registration info for attribute, a, for *new*
RegdAttr$_{new}$(a, el): Value of element corresponding to data element el, for attribute a in the set RegdAttr$_{new}$(a)
RegdAttr$_{base}$(a): Registration info for attribute, a, for *base*
RegdAttr$_{base}$(a, el): Value of element corresponding to data element el, for attribute a in the set RegdAttr$_{base}$(a)

Architectural Component Attribute Divergence:

Since reviewing divergence for individual attributes can be cumbersome, we bundle the divergence for attributes associated with a given architectural component. ArchCompAttrDiv(d) measures the diverge of all the attributes associated with a given architectural component. It is the average of the Attribute Divergences of all attributes associated with the architectural component. The Architectural Component Attribute Divergence value lies between zero and one and is calculated as in below

$$ArchCompAttrDiv(d) = \frac{1}{|ArchCompAttr(d)|} \sum_{a \in ArchCompAttr(d)} AttrDiv(a) \qquad (10)$$

Where
ArchCompAttr(d): Set of attributes associated with component d
AttrDiv(a): Attribute Divergence for attribute a

Behavioral Divergence

In our abstract model for architecture specification, the behavior of an architecture component is captured in terms of a state diagram. The behavioral specification is represented as the quintuple State, Trigger, Guard, Effect and Target. Each set of this quintuple is known as a behavioral unit in our model.

We evaluate the behavioral unit divergence between two versions *new* and *old* using an approach similar to that used for measuring attribute divergence. The divergence for each behavioral unit is captured by the Behavioral Unit Divergence or BehavUnitDiv(bu) as shown below.

$$BehavUnitDiv(bu) = \frac{1}{|BehavUnitEl(bu)|} \sum_{el \in BehavUnitEl(bu)} Abs[RegdBehavUnit_{new}(bu,el) - RegdBehavUnit_{base}(bu,el)] \quad (11)$$

Where
BehavUnitEl(bu): The set of elements associated with behavioral unit bu. This essentially consists of five elements State, Trigger, Guard, Effect and Target
$RegdBehavUnit_{new}(bu)$: Registration information for behavioral unit *bu* for the *new* version
$RegdBehavUnit_{new}(bu, el)$: Value of registration for element el, for behavioral unit bu in the set $RegdBehavUnit_{new}(bu)$
$RegdBehavUnit_{base}(bu)$: Registration information for behavioral unit *bu* for the *base* version
$RegdBehavUnit_{base}(bu, el)$: Value of registration for element el, for behavioral unit bu in the set $RegdBehavUnit_{base}(bu)$

The divergence associated with each behavioral unit is bundled into the Architecture Component Behavior Divergence, ArchCompBehavDiv(d), and is evaluated as in below. It is the average of behavioral unit divergence for all the behavioral units in the architectural component.

$$ArchCompBehavDiv(d) = \frac{1}{|ArchCompBehavUnit(d)|} \sum_{bu \in ArchCompBehavUnit(d)} BehavUnitDiv(bu) \quad (12)$$

Where
ArchCompBehavUnit(d): The set of behavioral units associated with the architecture component d
BehavUnitDiv(bu): Behavioral Unit Divergence for behavioral unit bu.

Just like the other divergence measures, the value of ArchCompBehavDiv(d) also lies between zero and one.

Architecture Component Divergence

While Architecture Component Service Divergence ArchCompSvDiv(d), Architecture Component Attribute Divergence ArchCompAttrDiv(d) and Architecture Component Behavioral Divergence ArchCompBehavDiv(d) capture the deviation of an architecture component with respect to the services, attributes and the behavior, Architecture Component Divergence captures the overall divergence of a component wrt the architecture specifications.

It is measured by the weighted average of the divergences of the services, the attributes and behavior where the weights correspond to the number of services, the number of attributes and the number of behavioral elements associated with the architectural component.
Architecture Component Divergence is evaluated as in below and its value lies between zero and one.

$$ArchCompDiv(d) = \frac{|ArchCompSv(d)| \times ArchCompSvDiv(d) + |ArchCompAttr(d)| \times ArchCompAttrDiv(d) + |ArchCompBehavUnit(d)| \times ArchCompBehavDiv(d)}{|ArchCompSv(d)| + |ArchCompAttr(d)| + |ArchCompBehavUnit(d)|} \quad (13)$$

Where
ArchCompSv(d): Set of services in component d.
ArchCompSvDiv(d): Architecture Component Service Divergence for component d
ArchCompAttr(d): Set of attributes in component d
ArchCompAttrDiv(d): Architecture Component Attribute Divergence for component d
ArchCompBehavUnit(d): Set of behavioral units for component d
ArchCompBehavUnitDiv(d): Architecture Component Behavior Divergence

Architecture Component Divergence also follows the relation $0 \leq ArchCompDiv(d) \leq 1$.

4.2 Structural Divergence

As explained previously, the *form* of an architecture is modeled using Form Units where an individual Form Unit is composed of one or many Form Unit Mapping(s). Each Form Unit Mapping is modeled by a pair of components, the cardinality associated with the components and the Rule that explains the relationship between the component pair.

Structural Divergence measures the deviation in the *form* of the architecture. To calculate the Structural Divergence, we first calculate the Form Unit Mapping Divergence *FormUnitMapDiv(fum)* for a given form unit *fum*. It is essentially the ratio of the change in the number of elements of the Form Unit mapping that a component is registered to, to the total number of elements in a given form unit mapping. The factor $RegdFormUnitMap_{base}(fum, el)$ represents whether the *base* component is registered to element *el* of form unit mapping *fum* or not. If it is, then the value is one, otherwise its value is zero. The factor $Abs[RegdFormUnit_{new}(fum, el) - RegdFormUnitMap_{base}(fum, el)]$ has a value of one if there is a change in the registration from the *base* version to the *new* version. (14)

$$FormUnitMapDiv(fum) = \frac{1}{|FormUnitMapEl(fum)|} \sum_{el \in FormUnitMapEl(fu)} Abs[RegdFormUnitMap_{new}(fum, el) - RegdFormUnitMap_{base}(fum, el)]$$

The overall Structural Divergence *StructuralDiv(a)* is measured by taking into account all the form unit mappings associated with a form unit and bundling it for all the form units associated with the architecture, a. The scope of the architecture could be restricted to an architectural region to derive the Structural Divergence for the region of interest.

We first derive the FormUnitDivergence(a,f) as

$$FormUnitDiv(a,f) = \frac{1}{|FormUnit(a,f)|} \sum_{fum \in FormUnit(a,f)} FormUnitMapDiv(a,f,fum) \quad (15)$$

And then calculate the Structural Divergence as

$$StructuralDiv(a) = \frac{1}{|ArchForm(a)|} \sum_{f \in ArchForm(a)} FormUnitDiv(a,f) \quad (16)$$

Where

FormUnitMapEl(fum): Set of elements for form unit mapping *fum*

RegdFormUnitMap$_{new}$(fum, el): Value of registration for element *el* in form unit mapping *fum* in the set for component *new*

RegdFormUnitMap$_{base}$(fum, el): Value of registration for element *el* in form unit mapping *fum* in the set for component *base*

ArchForm(a): Set of Form Units associated with architecture *a*

FormUnit(a,f): Set of Form Unit Mappings associated with form unit *f* in architecture *a*

FormUnitMapDiv(a, f, fum): Form Unit Mapping Divergence for form unit mapping *fum* for unit *f* in architecture *a*

The value of structural divergence lies between zero and one, with zero implying no change in the architectural form while one implies a complete overall of all the form unit mappings associated with the architecture.

5. Architectural Erosion

Architectural Erosion measures the loss of functionality or architectural form as a software system or component evolves. Analogous to the natural phenomenon of erosion, Architectural Erosion provides indictors to track loss of system functionality or structure.

Functional Erosion indictors focus on loss of system functionality while Structural Erosion indicators focus on the loss of architectural form.

5.1 Functional Erosion

Functional Erosion can be classified into Service Erosion and Attribute Erosion. Service Erosion focuses on the loss of system functionality or services supported while Attribute Erosion focuses on the loss of Data Attributes managed by the system.

Service Erosion

Service Erosion is essentially the ratio of the number of services lost to the total number of services originally supported. As noted in (17), we keep count of the original number of services in the variable *countServices* and the count of services lost in the *countLostServices* variable. *countServices* is incremented for each service that was originally supported i.e. when $RegdSvc_{base}(s) = 1$. On the other hand countLostServices is incremented when RegdSvc$_{new}$(s) is zero and RegdSvc$_{base}$(s) is one or when $RegdSvc_{new}(s) - RegdSvc_{base}(s) < 0$. Note that the value of RegdSvc$_{new}$(s) and RegdSvc$_{base}$(s) are either zero or one depending on whether the component *new* and *base* are registered to the service *s* or not.

$$\forall s \in ArchSvc(a)\ [(RegdSvc_{base}(s) = 1) \rightarrow countServices{+}{+};\ (RegdSvc_{new}(s) - RegdSvc_{base}(s) < 0) \rightarrow countLostServices{+}{+}] \quad (17)$$

The value of Service Erosion, SvcErosion(a), is obtained by dividing *countLostServices* by *countServices*

$$SvcErosion(a) = \frac{countLostServices}{countServices} \quad (18)$$

Service Erosion can be measured at any level of granularity, for an architecture as a whole or for a specific architectural region. It may comprehend one architectural element or multiple architectural elements. Assigning the appropriate set of services to the set ArchSvc(a) would determine the scope of this measure.

Attribute Erosion

Like Service Erosion, Attribute Erosion measures the loss of data managed by the system. The approach for computing the Attribute Erosion is exactly the same as Service Erosion. Two variables *countAttributes* and *countLostAttributes* keep track of the attributes originally supported i.e. $RegdAttr_{base}(a) = 1)$ and the attributes lost i.e. $(RegdAttr_{new}(a) - RegdAttr_{base}(a) < 0)$ respectively.

$$\forall a \in ArchSvc(ar)\ [(RegdAttr_{base}(a) = 1) \rightarrow countAttributes{+}{+};\ (RegdAttr_{new}(a) - RegdAttr_{base}(a) < 0) \rightarrow countLostAttributes{+}{+}] \quad (19)$$

Attribute Erosion, *AttrErosion(a)* is computed by diving *countLostAttributes* by *countAttributes.*

$$AttrErosion(a) = \frac{countLostAttributes}{countAttributes} \quad (20)$$

5.2 Structural Erosion

The evaluation of Structural Erosion is similar to the assessment of functional erosion. However in the case of Structural Erosion, we measure the loss of structural units from the architecture where structural unit is represented by form unit mappings.

Structural Erosion is evaluated as shown below

$$\forall fum \in ArchFormUnitMap(a)$$
$$[(RegdFormUnitMap_{base}(fum) = 1) \rightarrow$$
$$countFormUnit++; (RegdFormUnitMap_{new}(fum) \quad (21)$$
$$- RegdFormUnitMap_{base}(fum) < 0) \rightarrow$$
$$countLostFormUnit++]$$

In the above expression, we increment the *countFormUnit* for every form unit that the base component was registered to. The expression $(RegdFormUnitMap_{base}(fum) = 1)$ evaluates to true when the base component is registered to the form unit mapping *fum*. The variable *countLostFormUnit* keeps track of all the form unit mappings that are no longer supported by the *new* component though they were supported in the *base* component. Only then the term $RegdFormUnitMap_{new}(fum) - RegdFormUnitMap_{base}(fum) < 0)$ evaluate to true and *countLostFormUnit* incremented.

The Structural Erosion is computed by dividing *countLostFormUnit* by the *countFormUnit*.

$$StructuralErosion(a) = \frac{countLostFormUnit}{countFormUnit} \quad (22)$$

The structural erosion can be computed for either an architectural region or the architectural as a whole. A high value of *StructuralErosion(a)* signifies a significant loss in the structure of the architecture while a value of zero implies no loss in the architectural structure.

6. Evaluation

The proposed divergence and erosion measures were applied to a sample University Registration System where the architectural description of the baseline system consisted of 45 services and 22 data attributes distributed over 15 architectural components. The same system had 48 services and 29 data attributes distributed over 15 architectural components after a refactoring exercise.

One of the architectural components "Registration Manager" had 5 services and 3 attributes. We illustrate the application of the divergence measures using the Registration Manager component. For one of its service, "Drop a Class", the Input Event Divergence, IEDiv(s), was computed to be 0.33 using formula (1) as one of the three input events was dropped due to the elimination of a redundant UI screen. This service originally generated one output event "Recalculate Fee Bill". Post refactoring a new output event was added "Notify Instructor". As a result, the Output Event Divergence, OEDiv(s), was calculated as 0.50 using (2).

The service "Drop a Class" originally had two input data entities, "Course Info" and "Student Record." Post refactoring, the number of entities remained the same but an additional data element "Instructor Email Id" was added to the "Course Info" data entity, whereby the number of data elements associated with "Course Info" changed from 5 to 6. The number of data elements associated with "Student Record" did not change from the original list of 6. The Input Data Divergence, IDDiv(s) was thus calculated to be 0.083 using (3). There was no change in the one output data entity "Updated Student Record" for this service and so the Output Data Divergence was calculated as zero using (4).

"Drop a Class" had 2 pre-conditions "Student is registered for Class" and "Class has not been cancelled". A third pre-condition "Add/Drop Class Deadline Date has not passed" was added. Thus using (5), the Pre Conditions Divergence, PreCDiv(s), was calculated as 0.33. An additional post condition "Instructor has been notified" was added to the existing list of 3. So using formula (6), the value of Post Condition Divergence, PostCDiv(s), was calculated as 0.25.

The two Input Events were generated by 2 input screens and so, IEDep(s) equals 2, while OEDep(s) equals 2 as two services consumed the output events generated. The two input data entities "Course Info" and "Student Record" were created by two different services and so IDDep(s) also equals 2. Since three services consumed the one Output Data Entity, "Updated Student Record", ODDep(s) equals 3. The three pre-conditions were generated by 3 services while the four post-conditions were needed for 4 other service executions, thus PreCDep(s) equals 3 and PostCDep(s) equals 4. Using above info and formula (7), the service divergence for "Drop a Class" service was calculated to be 0.2385. This value of the Service Divergence would alert the System architect that some changes have happened wrt this service because of system evolution. So in case of any incompatibilities or functionally failures, this measure would highlight services for scrutiny.

The Service Divergence for the "Registration Manager" component was bundled to calculate the Architectural Service Divergence using formula (8) and its value worked out to 0.345. Similarly we computed the Architectural Component Attribute Divergence for the 3 attributes managed by the component using formulae (9) and (10) and it worked out to 0.25. Since Behavioral information was not available for this architecture, we could not compute the Behavioral Divergences.

The overall Architectural Component Divergence worked out to 0.32 using formula (13). This value was the highest among all the other architectural components. This implied that there was noticeable change in the implementation of the "Registration Manager" component post refactoring, and could therefore be a source of potential architectural mismatches.

It was interesting to note that for the University Registration System, the Structural Divergence calculated using formula (14), (15) and (16) worked out to zero. This would indicate that no changes were made to the relative positioning of the components in the architecture. Hence from the above information, the system architect would be able to conclude that the scope of the refactoring was localized to the enhancement of existing components and that no major architectural changes were done. The divergence measures, in general, are envisioned to provide guides to the system architect to identify architectural 'hotspots' for analysis to prevent undesired changes.

The Architectural Erosion measures were also exercised as part of our case study. The Service Erosion was found to be 0.063 using formula (17) and (18), while Attribute Erosion was 0.069 using (19) and (20). As would be obvious from the Divergence measures, the Structural Erosion indicator was evaluated to zero. The erosion measures are intended to alert the system architect when services, attributes or structural elements are dropped during system evolution. The System Architect should follow up to ensure that the loss does not impact the system negatively.

In summary, while the erosion metrics identify loss of functionality and architectural structure, the divergence measures used in conjunction with the erosion measures help to identify areas of activity (or change) in the architecture. The utility of these metrics is higher when software from different suppliers interoperate to deliver overall system functionality and the System Architect may not have detailed insight of all the supplier products.

7. Conclusion

In this paper we have developed a model for the architectural assessment of system evolution. We have also provided a vocabulary for the various aspects that are key to the measurement of system change. The architectural divergence and erosion indicators provide objective measures for the assessment of system change. It's worth mentioning that we have also developed a set of Architectural Drift metrics that leverage the Architectural Divergence indicators. However, we couldn't present those or a more detailed exposition of the evaluation done due to constraints of space.

Finally, even though there has been a fair amount of interest on system evolution, no research has proposed divergence or erosion indicators for an objective evaluation. From what we know, ours is the first attempt in that direction.

8. 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] Bowman, I. T., Holt, R. C., Brewster, N. V., "Linux as a Case Study: Its Extracted Software Architecture", *ICSE'99*, Los Angeles, CA, May 1999.

[3] Gall, H., Klosch, R., Mittermeir, R., "Object-Oriented Re-Architecting" *ESEC-5*, Berlin, Sep. 1995.

[4] Guo, G. Y., Atlee, J. M., Kazman, R., "A Software Architecture Reconstruction Method" *WICSA-1*, SA, 1999.

[5] Kazman, R., Carriere, J., "View Extraction and View Fusion in Architectural Understanding" *5th International Conference on Software Reuse*, Canada, 1998.

[6] Mikic-Rakic, M., Mehta, N. R., Medvidovic, N., "Architectural Style Requirements for Self-Healing Systems", *1st Workshop on Self-Healing Systems*, Charleston, Nov. 2002.

[7] Medvidovic, N., Egyed, A., Gruenbacher, P., "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery", 2nd International Workshop from Software Requirements to Architectures (STRAW), Portland, 2003

[8] O'Reilly C., Morrow, P., Bustard, D., "Lightweight Prevention of Architectural Erosion", 6th International Workshop on Principles of Software Evolution, 2003

[9] Bhattacharya, S., Perry, D. E.,. "Contextual Reusability Metrics for Event-Based Architectures", 4th ACM-IEEE International Symposium on Empirical Software Engineering, November 2005, Australia

[10] van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N., "Taming Architectural Evolution", 8th European Software Engineering Conference, 2001, 1-10

[11] Bass, L., Clements, P., Kazman, R., "Software Architecture in Practice", Addison Wesley, 1999

[12] Fenton, N., Pfleeger, S., "Software Metrics: A Rigorous and Practical Approach", PWS Publishing Company, 1997

[13] Shaw, M., Garlan, D., "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall, 1996

[14] Bosch, J, "Design & Use of Software Architectures: Adopting and evolving a product-line approach," Addison-Wesley, 2000

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