A Metrics Suite for Version Management

D. Janakiram and A. Ananda Rao,
Distributed & Object Systems Lab
Dept of Computer Science & Engg.,
Indian Institute of Technology, Madras, India
{djram,anand}@dos.cs.iitm.ernet.in

Abstract

In existing version management systems, different versions of an artifact are distinguished between the designs that are same and that are different based on various types of attributes of an artifact. The attributes[1] that govern the design of an artifact are called key attributes (e.g. interface). The attributes that do not govern the design of an artifact are called non-key attributes (descriptive attributes). Changes in key attributes lead to new design version of an artifact and changes in non-key attributes lead to design equivalents. In present models, designer is responsible for categorizing the artifacts' attributes into key and non-key attributes. More over the division is performed conceptually. Therefore this paper is aimed at proposing a metrics suite that can be used to categorize the attributes as key and non-key attributes. This metrics suite is developed based on the design complexities of the attributes of an artifact. The proposed metrics have been validated formally using Weyuker's principles. Empirical validation has been provided by taking different versions of Restricted Focus Viewer (RFV) software as a case study.

Keywords: complexity, design, key attributes, non-key attributes, metrics, version management.

I. INTRODUCTION

The software systems constantly evolve because of the continuous change in requirements. That is why in software development, large projects are generally implemented based on iterative paradigm. Iterative process provides successive refinements over previous iterations. These refinements of a project are managed by maintaining different configurations of the various artifacts.

Semantics based version management in databases is proposed in [1]. In this, properties of an artifact are categorized into two different types based on the semantics of an artifact as design attributes (key attributes) and non-design attributes (non-key attributes). The design attributes govern the design of an artifact while non-design attributes do not. If any change occurs in the design attributes, it leads to creation of new design version of an artifact. On the other hand changes in non-design attributes lead to design equivalents. So, classifying the attributes into design and non-design is of utmost importance.

However, this categorization of attributes is done by the designer conceptually or intuitively in above model. Hence the division of artifact's attributes into different types depends on the experience (knowledge) of the designer. No scientific way is provided in the literature for the division of the artifact's attributes. To this end, this paper aims at providing a scientific basis by proposing a metrics suite. Based on this metrics suite, artifact attributes (instance variables and methods) are classified as design attributes and non-design attributes. These metrics are developed based on the design complexities of artifact attributes. The reason for considering design complexity is that software maintenance depends on software design whose major concern is changeability [2]. Therefore changeability is correlated with the complexity at design level. Secondly, the proposed metrics are validated formally against the Weyuker's principles [3]. All applicable principles are satisfied by the proposed metrics. Further an empirical validation has been conducted by taking different versions of Restricted Focus Viewer (RFV) software as a case study [4]. In the process of empirical validation, Unified Representation of Artifacts (URA) model [5] is used to represent the different artifacts of software system under consideration.

Organization of the paper is as follows. Section 2 presents the related work on version management and metrics. A brief introduction of the URA (artifacts' representation) model is presented in section 3. It also explains version management concepts briefly. Section 4 addresses the proposed metrics with an example. In section 5, the proposed metrics are validated analytically and empirically. Section 6 concludes the paper and gives directions for future work.

II. RELATED WORK

Version management of composite objects in CAD databases is explained in [6]. This work distinguishes when two instances of the same type are different objects and when there are merely different versions of the same object. The key to deciding this issue is to view each attribute of a design object as either intrinsic or non-intrinsic. If intrinsic attributes change so does the object. Non-intrinsic properties on the other hand, can be modified without changing the object in any significant way. Interface is an example of intrinsic attributes, whereas properties such as "name of the artifact" and "designer" etc. are examples of non-intrinsic attributes. Management of different design versions of an artifact in design databases is discussed in [1]. In this model, attributes of an artifact are divided based on semantics into two different types as discussed in previous section. For example, consider the design of an elastically stressed element. the attributes considered include Young's modulus, thermal conductivity, ductility, Poisson's ratio, colour etc. Of all these attributes, only Young's modulus govern the design and can be considered as a key attribute. Key attributes are also called as versioning or design attributes in the paper. Any change in

[1] In the paper attribute means feature of an artifact

Young's modulus attribute can be considered to be a new design of an elastically stressed element. However, changes in other attributes such as colour, ductility etc. will not create new design of an elastically stressed element. These attributes are called non-key attributes (non-versioning attributes).

A generic model for semantics based versioning in projects is proposed in [7]. It addresses the changes as well as the change propagations. The model is developed based on the URA mechanism. The next subsection gives the brief introduction about the generic model. The authors also proposed version management in unified modeling language [8]. This paper discusses the semantic based version management in the projects that are represented by UML. The UML class diagrams are used to represent the semantic entities in the project. The version propagation is captured through class diagrams and their relationships. In all the above said models, categorization is done by the designer conceptually or intuitively.

Majority of the object oriented metrics are proposed to evaluate cohesion and coupling of artifacts in the software systems. Most popular among the metrics suites are the ones proposed by the Chidamber and Kemerer widely known as CK metrics [9]. Even though these metrics are defined at design level these are not suitable for Version Management (VM). For example, Weighted Method per Class (WMC) metric is not defined specifically and it is left as an implementation decision. More over it does not take the advantage of some of the additional information available in OO design. For example, the concept of interface size gives a measure of the means for information to flow in and out of a class. Its definition does not include the use of interface size. Since the design complexities of different types of variables (that are independent of language) are available (refer table 1), it is easy to compute design complexities of various methods. Therefore WMC metric is not useful for version management. Similarly the metrics Depth of Inheritance Tree (DIT), Number of Children (NOC) etc. that are developed by CK are not suitable for VM as their purpose is different. Eder et al define a frame work aimed at providing qualitative criteria for cohesion in [10]. These metrics are not suitable for version management as they are represented qualitatively. Hence separate set of metrics are proposed for VM.

III. THE URA MODEL

In this section, initially the representation model called URA is explained. Next, semantics based version management using URA graph is presented briefly. At the end object oriented software system design is defined formally.

A. Introduction to URA

A meta level entity called URA which represents an artifact of any type or granularity is presented in [5] [7]. Any logical entity of interest is an artifact. Artifacts map to physical entities in different ways like classes, sets of classes, subsystems, documents etc. The structure of URA is shown in figure 1. A URA mainly consists of three components - first one for extracting the artifact from the information system, second one contains the information about the artifact, third and the last one enforces authentication mechanisms. A set of features is associated with the URA, which allows it to be classified and queried. These

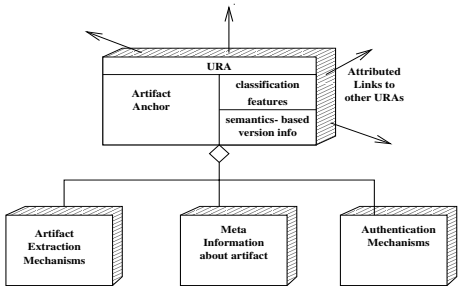

Fig. 1. Structure of a URA

features can be either attributes or functionalities of the artifact. Semantics based version information set keeps track of the evolution of the artifact. In addition to these, there are labeled links pointing to other URAs, which reflect the relationship between the artifacts that the URAs represent.

A project is represented as a directed graph of URAs. The graph will evolve as changes occur in the project. An artifact in the project is represented as a URA, that is a node in the URA graph. Directed edges in the graph are labeled. These labels are the relationships between the artifacts. The labeled links indicate the dependencies between the nodes of the graph and the need to propagate the changes. A pivot node in the graph represents the whole project. URA nodes are linked to pivot node by dependency links. Changes are propagated to this node too. The version of this node is the version of the project.

B. Version Management Concepts

In OO systems class is considered as a basic unit. Therefore each class is treated as an artifact and represented as a URA. Links between the URAs depict the relationship between the classes such as aggregation, association and inheritance. In a URA graph it is easy to manage the versions. There are two basic issues of version management. These are version change and version propagation. Version change of an artifact can occur through versioning (key) or non-versioning (non-key) attributes. The change in an artifact causes change in the other related artifacts. The change is propagated to related artifacts based on *Focus* (cohesion) and *Cdegree* (coupling) values. The *Focus* is nothing but the probability that the change does not necessitate similar changes in the other related artifacts and the *Cdegree* is the indicator of an amount of dependency that exist between two related artifacts. The value of the *Cdegree* has a range [0,1]. If the *Cdegree* value is more than the threshold (say 0.5) then the link is said to be strong and called as *cohesive* link. Otherwise, if the value is less than the threshold then the link is weak and called a *non-cohesive* link. If *Focus* value is more, then the probability of propagating changes to other related artifacts is low. Similarly if *Cdegree* value is more, then the changes in the artifact should propagate to the other related artifacts. Some of rules that are used to propagate change in the artifact to related artifacts are as follows.

1) If *Focus* is less than the *Cdegree* value then propagate changes to other related artifacts.

2) If *Focus* is greater than the *Cdegree* value then the change is local to the artifact and need not propagate to other related artifacts.

The complete set of rules can be found in [7]. Since class is considered as a basic unit, It is obvious that if an attribute is a private attribute of a class then the changes of that attribute may not necessitate changes in related classes. Thus, the *Focus* of this attribute is high. Similarly, public attributes *Focus* is low. In case of protected attribute the change may affect the related artifacts depending on link between the URAs. If the link is inheritance link then the *Focus* is low, otherwise *Focus* is high. The inheritance link is considered as *cohesive* link because changes made to base class affect the derived class. Similarly aggregation relation is also considered as a *cohesive* link because changes made to part class affect the whole class. Association link can be considered as *cohesive* or *non-cohesive* depending on the amount of the association between related classes.

C. Formal Definition OO System Design

Object oriented design, D, can be conceptualized as a relational system that consists of classes/objects as elements, empirical relations and binary operations that can be performed on these elements. Notationally it is defined as follows.
D = (E, $R_1...R_n$, $O_1..O_m$) Where E is set of class/objects, $R_1...R_n$ empirical relations on E and $O_1..O_m$ are binary operations on E.

To be able to measure some thing about an object design the system defined above needs to be transformed to a formal relational system. The formal relation system, F, be defined as follows.
F = (A, $X_1...X_n, Y_1...Y_m$) where A is a set of elements, $X_1...X_n$ are formal relations on A (e.g., >,<,=) and $Y_1...Y_m$ are binary operations on A (e.g., +,-,*). Hence system is defined formally.

IV. VERSION MANAGEMENT METRICS

In this section, metrics proposed for version management are discussed. First the metrics that are applicable for categorization of instance variables are presented. Next, metrics that are used to categorize methods of an artifact are discussed. Class is considered as a basic unit while developing metrics.

A. Metrics for Variable Categorization

There are four metrics developed for variable categorization based on design complexities of variables. These metrics are developed based on the interactions of the variables with the methods within the class and out side the class, type (size) of the variable and scope of the variable. A variable is considered as more complex (versioning variable) if it is complex (larger) in size, has more interactions within and outside a class. Larger the interactions of a variable results in more maintainability if it is changed. The proposed metrics are discussed below.

A.1 Interaction Metric Inside the Class (IM_i)

This metric quantifies the interactions inside the class. It is defined as the ratio of number of methods inside the class that reference the variable to the total number of methods in a class.

$$IM_i = NM_{irv}/TM$$

where NM_{irv} is number of methods inside the class that reference the variable and TM is total number of methods in the class

If all the methods in a class reference the variable, then IM_i is maximum i.e., 1 and if non of the methods in a class reference the variable the value of IM_i is minimum i.e., 0. Consider the interactions shown in figure 2(a) that explains the computation of metric IM_i. In the figure variable a_1 is referenced by all the four methods x_1 to x_4. So the value of IM_i for the variable a_1 is 4/4 = 1. The variables a_3 and a_5 are referenced by only one method and the value of IM_i is 1/4 = 0.25. Similarly the values of IM_i for other variables can be computed. Note that in this paper all the proposed metrics values are given as normalized values i.e., their values range from 0(zero) to 1(one).

A.2 Interaction Metric Outside the Class (IM_o)

This metric quantifies the interactions outside the class. It is defined as the ratio of the number of methods outside the class that reference the particular variable to the total number of methods in the referenced classes.

$$IM_o = \frac{\sum_{i=1}^{N-1}\sum_{j=1}^{M} V_{ij}}{\sum_{i=1}^{N-1}\sum_{j=1}^{M} M_{ij}}$$

where V_{ij} = 1 if j^{th} method of i^{th} class references the variable under consideration and V_{ij} = 0 otherwise. M_{ij} = 1 for every j^{th} method of i^{th} class, N represents number of referenced classes including the one that is under consideration and M represents number of methods for each class. For example, if system contains four classes that are to connected each other then after considering one of the classes for metric calculation remaining are three.

The values 1 and 0 for V_{ij} and M_{ij} are chosen for convenience and normalization. For explaining the computation of metric IM_o the interactions that are shown in figure 2(b) are considered. In that figure class B and class C are out side related classes for class A. The class B is having 3 methods and class C is having 4 methods. So the total number of methods out side the class A are 7. The variable a_1 is referenced by three methods and IM_o metric value for variable a_1 is 3/7 = 0.43. In a similar way IM_o values for other variables can be computed.

A.3 Variable Type Metric (VTM)

It is defined as the ratio of size of the type of a variable to the maximum size. The maximum size is the size of file type variable (see table I). The size of a variable or parameter is a specified constant, specifying the complexity of the variable type. By using this metric we can measure how complex a variable is. The size values for different variables are taken from the table I that are suggested in [2], [11] and [12]. These values are given based on the design complexities of various variables by the experienced designers. Formally it is defined as follows.
VTM = S / M_s where S is the size of the variable type and M_s is the maximum size of variable type. For example, if the type of variable V is integer then the type metric VTM for variable V is 1/10 = 0.1. Similarly the type metric VTM value for variable file a_1 in the above example class A is 10/10 = 1.

A.4 Scope Metric (SM)

This gives the weight of public variables in a class. Changing the scope of a variable to public will not have effect, but

TABLE I

SIZE OR COMPLEXITY OF VARIABLES TYPES

Type	Size
Boolean	0
character or Integer	1
Real, Float	2
Array	3
Pointer	5
Record, Struct, Object	6
File	10

TABLE II

CALCULATED VALUES FOR THE METRICS

	a_1	a_2	a_3	a_4	a_5	a_6
IM_i	1	0.5	0.25	0.0	0.25	0.5
IM_o	0.43	0.0	0.0	0.0	0.71	0.71
VTM	1	1	1	0.2	0.1	0.1
SM	0.66	0.66	0.66	0.66	0.33	0.33

changing the public variable to non-public restrict visibility and there is an impact in classes which are in association with the changed class i.e., referring the variable. To calculate the scope metric, the number of public and non-public variables within the class are counted. To make the metrics more general and non-language specific, the variables are divided into two parts as public and non-public.

The scope metric for public variables is defined as the ratio of number of public variables to the total number of variables within the class.
SM = NV_{pub} / TV, where NV_{pub} is number of public variables in a class and TV is total number of variables in a class. Similarly, the scope metric for a non-public variables can be defined as ratio of non-public variables to the total number of variables within the class.

All the above mentioned metrics are calculated for figures 2(a), (b) and values are tabulated in table II. The variable is categorized as versioning attribute if majority of the metrics (say fifty percent i.e., two metrics out of four) produces the greater or equal value than the threshold (say 0.5) value. Otherwise the attribute is treated as non-versioning attribute. From the table II, attributes a_1, a_2, a_3 and a_6 are categorized as versioning attributes. Others are categorized as non-versioning attributes. If the criteria is such that more than fifty percent of metrics have value that is more or equal to the threshold value, then the attribute a_1 alone is considered as versioning attribute because IM_i, VTM and SM metrics values are more than threshold value. The rest of the attributes are categorizes as non-versioning attributes. However it is designer's choice to choose one among the above mentioned methods to categorize the attributes as he/she is having the knowledge about the system.

B. Metrics for Method Categorization

These metrics are developed based on the interactions with the variables and methods within and outside the class, size of the methods, return type and scope of the methods. A metric is proposed for each characteristic. These metrics values are used to classify whether a particular method is versioning or non-versioning. The metrics are explained as follows.

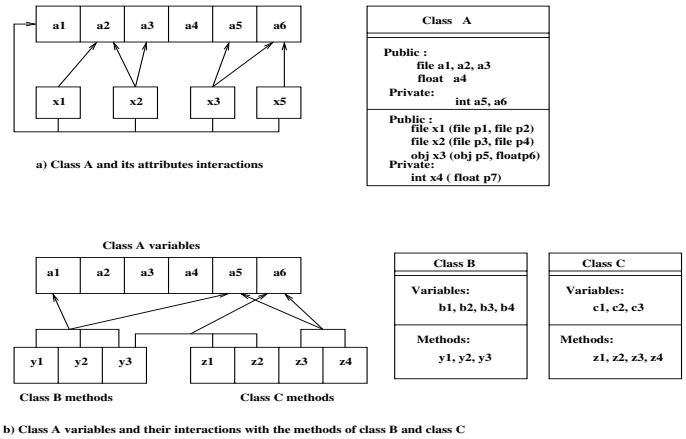

Fig. 2. Example class diagrams

B.1 Method Invocation Metric Inside the Class (MI_i)

It is defined as the ratio of the number of methods within the class that invoked the method under consideration to the total number of methods in the class.

MI_i = MI / TM-1; where MI is the number of methods that invoked the method under consideration and TM is total number of methods in a class.

The metric MI_i measures the invocations within the class. MI_i is 1 (maximum) if the method under consideration is invoked by all the methods in the class. If none of the methods invokes the method under consideration then MI_i value is 0 (minimum). For explaining the computation of MI_i metric, consider the invocations of example class A shown in figure 3(a). In figure, the method x_1 is invoked by all the methods and so MI_i is 3/3 = 1. Similarly MI_i metric values for all the methods are calculated and tabulated in table III.

B.2 Method Invocation Metric Outside the Class (MI_o)

It is defined as the ratio of the number of methods outside the class that invoked the method under consideration to the total number of methods in the referenced classes.

$$MI_o = \frac{\sum_{i=1}^{N-1}\sum_{j=1}^{M} m_{ij}}{\sum_{i=1}^{N-1}\sum_{j=1}^{M} M_{ij}}$$

where m_{ij} = 1 if j^{th} method of i^{th} class references the variable under consideration and m_{ij} = 0 otherwise. M_{ij} = 1 for every j^{th} method of i^{th} class.

For explaining the computation of MI_o metric, consider the figure 3(b). In figure, y_1, y_2 and y_3 are the methods in class B and z_1 to z_4 are the methods in class C. The method x_1 is invoked by 5 methods and total number of methods in outside referenced classes is 7. So MI_o for method x_1 is 5/7 = 0.71. Similarly the MI_o for the other methods is calculated and tabulated in table III.

B.3 Method Variable Reference Metric Inside the Class (MVR_i)

It is defined as the ratio of number of variables within the class referred by the method to the total number of variables in the class.

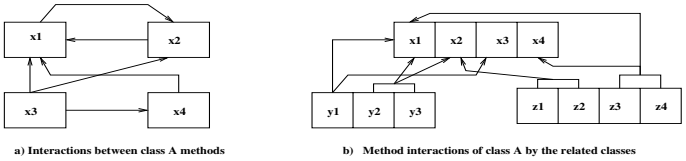

Fig. 3. Example diagrams of classes A,B,C and their method interactions

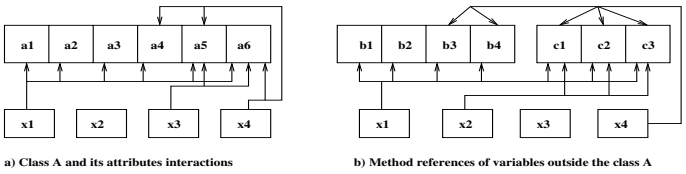

Fig. 4. Example class diagrams and their variable interactions

MVR_i = NM_v / T_v where NM_v is the number of variables in the class that are referenced by the method under consideration and T_v is the total number of variables in the class. In figure 4(a), the method x_1 references all the six variables, therefore the metric MVR_v is 1. Similarly MVR_i values are calculated for all the methods in figure and tabulated in table III.

B.4 Method Variable Reference Metric Outside the Class (MVR_o)

Measures the variable references outside the class. It is defined as the ratio of number of variables outside the related classes referred by the method to the total number of outside variables in referenced classes.

$$MVR_o = \frac{\sum_{i=1}^{N-1}\sum_{j=1}^{M} m_{ij}}{\sum_{i=1}^{N-1}\sum_{j=1}^{M} M_{ij}}$$

where m_{ij} = 1 if j^{th} method of i^{th} class references the variable under consideration and m_{ij} = 0 otherwise. M_{ij} = 1 for j^{th} method of i^{th} class.

Attributes and parameters have types which contribute to coupling. In figure 4(b), method x_1 references all the eight outside variables and MVR_o value is 7/7 = 1. Similarly MVR_o values are calculated for all the methods in the figure and tabulated in table III.

B.5 Parameter Type Metric (PTM)

The parameter type metric is computed using the size of the parameter types given in the table I. More the number of parameters passed, stronger the coupling. That is if method produces more parameters then it is more complex. It is defined as ratio of the sum of sizes of the parameters to the product of the number of parameters and the maximum size (file size which has a value 10 in the table I).

$$PTM = \frac{\sum_{i=1}^{n} P_i}{n * Max.Size}$$

where n = number of parameters, P_i = the size of the i^{th} parameter and Max.Size is the maximum size of the parameters (i.e., file size 10). In the example class A, the method x_1 has two parameters of type file whose size is 10. The maximum size is 10 and so the PTM metric value for the method x_1 is (10+10)/2*10 = 1. Similarly PTM metric values for all the methods are computed and tabulated in table III.

TABLE III
CALCULATED VALUES FOR THE METRICS

	X1	X2	X3	X4
PTM	1	1	0.4	0.2
MI_i	1	0.66	0.0	0.33
MI_o	0.71	0.57	0.14	0.3
RTM	1	1	0.6	0.1
MVR_i	1	0.0	0.33	0.5
MVR_o	1	0.43	0.0	0.71

This metric along with MI_i metric seems to be closely related with the ICH (information based flow cohesion) metric defined by Lee et al [13]. But ICH is defined by the methods that are implemented within a class under consideration. Therefore ICH is not applicable for version management, because it does not consider inherited attributes of a class. Moreover, ICH considers number of parameters but not the strength of these parameters. Since the design complexities (table I) are available for various type of attributes the paper considers not only number of parameters but also the strength of those parameters. However, the numerator of PTM metric is similar to Operation Argument Complexity metric mentioned in [2].

B.6 Return Type Metric (RTM)

Whenever a method is invoked, its parameters are used for some internal computation and a value may be passed back to the calling routine. The size of return value is treated same as the sizes of the other parameters and values are taken from table I. The return type metric is defined as the ratio of the size of the method's return type to the maximum size of the return type.

RTM = rs / rmax, where rs is the size of the return type and rmax is the maximum size of the return type. In the example class A (figure 2), the methods x_1 and x_2 having file as the return type. The maximum return type is 10 and so RTM metric value for methods x_1 and x_2 is 10/10 = 1. Similarly scope metric can also be defined for methods as in the case of instance variables.

All the metrics are calculated for class A and tabulated in table III. Methods are categorized as versioning and non-versioning attributes as in the case of variables. If the criteria is such that the more than fifty percent of the metrics have value that is equal or greater the threshold value, then from the table methods x_1 and x_2 are categorized as versioning methods and rest of the methods are categorized as non-versioning methods.

V. EVALUATION OF PROPOSED METRICS

In general, there are two ways of evaluating a metric: formally and empirically [14]. Formal evaluation is carried out first and next empirical validation is performed.

A. Formal Validation of Metrics Using Weyuker's Properties

Generally formal evaluation will be performed by evaluating metrics against the set of Weyuker's proposed principles [3]. Weyuker has developed a formal set of desiderata for software metrics and has evaluated a number of existing software metrics using these properties. These desiderata include notion of

TABLE IV
METRICS EVALUATION TABLE FOR VARIABLES

Metric	P1	P2	P3	P4	P5	P6	P7	P8	P9
Scope	Y	Y	Y	Y	NA	Y	NA	Y	NA
VTM	Y	Y	Y	Y	NA	N	NA	Y	NA
IM_i	Y	Y	Y	Y	NA	Y	N	Y	N
IM_o	Y	Y	Y	Y	NA	Y	N	Y	N

TABLE V
METRICS EVALUATION TABLE FOR METHODS

Metric	P1	P2	P3	P4	P5	P6	P7	P8	P9
Scope	Y	Y	Y	Y	NA	Y	NA	Y	NA
PTM	Y	Y	Y	Y	NA	N	NA	Y	NA
RTM	Y	Y	Y	Y	NA	N	NA	Y	NA
MI_i	Y	Y	Y	Y	NA	Y	N	Y	N
MI_o	Y	Y	Y	Y	NA	Y	N	Y	N
MVR_i	Y	Y	Y	Y	NA	Y	N	Y	N
MVR_o	Y	Y	Y	Y	NA	Y	N	Y	N

monotonicity, interaction, non-coarseness, non-uniqueness and permutation. Even though all proposed metrics are validated against all the Weyuker's principles, because of space constraint, only one principle is presented as an example. However, two tables are presented which show the metrics that satisfy the corresponding property. Table IV shows metrics that corresponds to variable categorization and table V represents metrics that corresponds to method categorization.

Property : Non -Coarseness

Given a class A and a metric μ, another class B can always be found such that $\mu(A) \neq \mu(B)$ (μ represents a metric). This states that not every class can have the same value for a metric. The metric should not have the same value for all entities, otherwise it has lost its value as a measurement.

Example 1: VTM metric considers the sizes of different types of variables. This metric satisfies above property. Consider two variables, int v_1 and float v_2. Then $\mu(v_1)$=1/10=0.1 and $\mu(v_2)$=2/10=0.2. Therefore $\mu(v_1) \neq \mu(v_2)$. Note that the symbol "μ" represents corresponding metric in the discussion.

Example 2: MI_i metric measures number of method invocations within the class. It satisfies the above property. Consider a class in which method x_1 is invoked by 3 methods out of total of 5 methods, and method $x_2$is invoked by 2 methods out of total of 5 methods then $\mu(x_1)$ = 3/5 = 0.6 and $\mu(x_2)$ = 2/5 = 0.4. Therefore $\mu(x_1) \neq \mu(x_2)$. Similarly, all the proposed metrics are validated against all principles.

Tables IV and V show the consolidated evaluation of metrics using Weyuker's Properties for all the proposed metrics. In both the tables Y represents satisfied, N represents not satisfied and NA represents not applicable. From the tables, it is clear that the metrics proposed for the methods satisfy all the applicable Weyuker properties. Hence proposed metrics are validated analytically.

B. Empirical Validation and Case Study

Restricted Focus Viewer (RFV) software has been chosen for carrying out the empirical validation and case study [4]. There are two versions of the RFV. Version 2.1 is the latest version which contains many new features and is more flexible for experiment. Version 1.1 is the old version and has been replaced by version 2.1. Version 1.1 has 13 classes and version 2.1 has 35 classes. Major changes have been incorporated in version 2.1 and this is evident from 22 new classes added. The empirical validation is carried out in the following steps.

- First, the source code is reverse engineered into UML class diagrams. Rational Rose software has been used for this purpose [15]. The simplified class diagrams for version 1.1 and 2.1 are shown in figures[2] 5(a) and 6 respectively.
- In the simplified UML diagrams various links like inheritance etc are shown symbolically.
- The values for the proposed metrics are calculated for the classes that exist in version 2.1.
- From the above metrics values, attributes of different classes and classes themselves are classified into versioning and non-versioning. Classes that produce more than fifty percent of its attributes as versioning attributes are considered as versioning classes.
- UML class diagrams are converted to URA graphs. Simplified version of URA graphs for version 1.1 and version 2.1 is shown in figures[3] 5(b) and 7 respectively.
- Whenever there is a change in the system, based on the type of the attribute in which the change is occurred, the scope of the attribute and the link that exist between the artifacts, the change is propagated to related artifacts.
- The new URA graph figure 7 represents the evolved version of the system.

The work related to empirical validation is the calculation of metrics and classifying the attributes and classes into versioning and non-versioning. The metric values for classes in version 2.1 are calculated and one among them is tabulated in tables VI and VII. From the metric values, attributes and classes are classified into versioning and non-versioning. Though the metrics are calculated for all the classes, for the purpose of this example, only one table is shown with respect to class RFV-Text-Line. The version of RFV is changed from version 1.1 to 2.1 due to the following reasons.

From the tables, it found that the class RFV-Text-Line is a versioning class. Out of 9 metrics, 5 metrics are having more

TABLE VI
METRICS CALCULATION FOR VARIABLES OF RFV-TEXT-LINE

Metrics	Label	font	Font-metrics	Color	Height-offset
IM_i	0.5	0.5	0.5	0.5	1
IM_o	0	0	0	0	0
VTM	0.3	0.6	0.6	0.6	0.1
SM	1	1	1	1	1

TABLE VII
METRICS CALCULATION FOR METHODS OF RFV-TEXT-LINE

Metrics	MI_i	MI_o	MVR_i	MVR_o	PTM	RTM	SM
Calculate-Textsize	0	0	0.4	0	0.4	0	0.5
Draw	0	0	0.8	0	0.6	0	0.5

[2]Figures contain only aggregation relationships i.e., any change to part affect the whole

[3]Filled circle in the figure indicates that the link is cohesive link i.e., strong coupling exists between the nodes.

than threshold (0.5) value. As more than 50 percent of metrics of class RFV-Text-Line is versioning, it is classified as versioning class. This is a new class introduced in version 2.1 and change propagation as a result of the introduction of new versioning class is shown in figure 7. Initially all the classes in version 1.1 including the pivot class RFV are assigned version 1.1 (newly added classes are not give version numbers). The version change is propagated to the pivot class RFV through class RFV-Input-File. Since whole-part relation exist between RFV-Text-Line and RFV-Input-File, version change is propagated to RFV-Input-File. Similarly, version change is propagated from RFV-Input-File to RFV because whole-part relation exist between them. Hence version of RFV is changed from 1.1 to 2.1.

There are number of other classes and attributes which are found to be versioning. One among them is RFV-Input-File class in version 1.1. For this class two versioning methods getFileIndex and setFileIndex are added in version 2.1. After calculation of metrics these are found as versioning methods. Therefore, addition of these methods to RFV-Input-File class in version 1.1 evolves[4] into its new design. Since this class has a cohesive link (whole-part relation) directly to RFV, the evolution is propagated to RFV and version of RFV is changed from 1.1 to 2.1. Hence, finally the pivot node class RFV represents the latest version. Thus the version of software RFV is changed from 1.1 to 2.1 because versioning classes and versioning attributes are added to version 1.1. For this example, no contradiction was found, that is the metrics say version but actual system did not version it. Thus the proposed metrics are applied on different versions of real software system RFV and satisfactory results are obtained which validates the proposed metrics empirically.

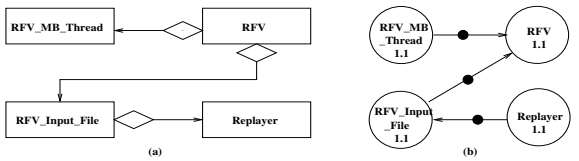

Fig. 5. Simplified class diagram and its URA diagram for RFV 1.1

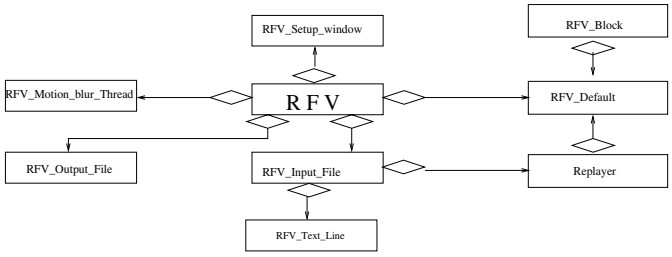

Fig. 6. Simplified class diagram of RFV 2.1

VI. CONCLUSIONS AND FUTURE DIRECTIONS

This paper proposed metrics suite for attributes categorization in version management. Using this metrics suite, attributes of an artifact are classified into two types as key and non-key attributes. Metrics suite is proposed based on the properties,

[4]Adding or changing any type of attribute, the artifact evolves into its corresponding version.

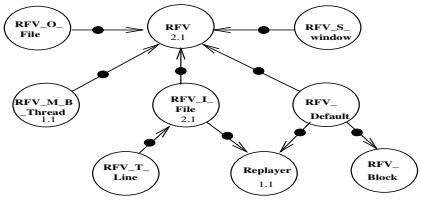

Fig. 7. URA diagrams of RFV 2.1

design complexities and information flow of the artifacts. The proposed metrics are validated formally using Weyuker's principles and found that all applicable principles are satisfied. The metrics have also been empirically validated using various versions of RFV software as a case study. Therefore, the proposed metrics can be used for automatic categorization of attributes into versioning and non-versioning.

Automation can be done for carrying out empirical validation on large systems. These metrics can also be extended to different purposes such as effort estimation and maintenance effort.

REFERENCES

[1] Ramakrishnan. R. and D. Janaki Ram., *Modeling Design Versions*, Proceedings of 22^{nd} International Conference on VLDB, Mumbai(Bombay), India, pp. 556-566, September 1996.

[2] Rajendra K.B. and Vaishnavi K.V., *Predicting Maintenance Performance Using Object Oriented Design Complexity Metrics*, IEEE Transactions on Software Engineering, Vol. 29, No. 1, January 2003.

[3] Weyuker E.J., *Evaluating Software Complexity Metrics*, IEEE Transactions on Software Engineering, Vol. 14, pp.1357-1365, Sep. 1988.

[4] Restricted Focus Viewer Website, Maintained by Anthony R. Janasen, Available at $http://www.csse.monash.edu.au/tonyj/RFV$.

[5] Srinath. S., *URA: A Paradigm for Context Sensitive Reuse*, A Thesis of Master of Science by Research, Department of computer Science & Engineering, Indian Institute of Technology, India, April 1998.

[6] Rafi Ahmed and Navathe. S.B., *Version Management of Composite Objects in CAD Databases* Proceedings of ACM SIGMOD international conference on Management of data, pp. 218 - 227, Denver, Colorado, May 29-31, 1991.

[7] Srinath. S., R. Ramakrishna and D. Janaki Ram, *A Generic Model for Semantics Based Versioning in Projects*, IEEE Transactions on Systems, Man and Cybernetics, Part A, Vol. 30, No. 2, pp. 108-123, March 2000.

[8] Janaki Ram. D., M. Sreekanth and A. Ananda Rao, *Version Management in Unified Modeling Language*, Proceedings of 6^{th} International Conference on Object Oriented Information Systems (OOIS 2000), London, pp. 238-252, December 18-20, 2000.

[9] Chidember S.R. and Kemerer C.F., *A metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, Vol. 20, No. 6, June 1994.

[10] Eder J. Kappel G. and Schrefl M., *Coupling and Cohesion in Object-Oriented Systems*, Technical Report, University of klagenfurt, 1994.

[11] Abbott D., *A Design Complexity Metric for Object-Oriented Development*, Masters thesis, Dept. of Computer Science, Clemson Univ., 1993.

[12] Chen J-Y and Lu J-F., *A New Metric for Object-Oriented Design*, Information and Software Technology, pp. 232-240, April 1993.

[13] Lee Y.-S., Liang B.-S, Wu S.-F., and Wang F.-J, *Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow*, In Proc. International Conference on Software Quality, Maribor, Slovenia, 1995.

[14] Cahnge C.K and Jane C.H., *Measuring the Intensity of Object Coupling in C++ Programs*,IEEE Conference, 2001.

[15] *Rational Rose-The visual modeling tool, Rational Software Corporation*, Available at $< http://www.rational.com/products/rose/index.jsp >$.