A FORMEL LANGUAGE FOR SOFTWARE REUSE

ZINA HOUHAMDI

Computer Science Department, University of Biskra
BP 145, Biskra RP, 07000, Algeria.
E-mail: z_houhamdi@yahoo.fr

Abstract : Software reuse has been claimed to be one of the most promising approaches to enhance programmer productivity and software quality. One of the problems to be addresses to achieve high software reuse is organizing databases of software experience, in which information on software products and processes is stored and organized to enhance reuse. This paper presents a new approach to define and construct such databases called the Reuse Description Formalism (RDF). The formalism is a generalization of the faceted index approach to classification. Unlike the faceted approach, objects in RDF can be described in terms of different sets of faceted and in terms of other object descriptions. This allows a software library to contain different classes of objects, to represent various types of relations among these classes, and to refine classification schemes by adding more detail supporting a growing application domain and reducing the impact of initial domain analysis. In particular, *RDF* provides a specification language based on concepts of set theory capable of representing a rich variety of software and non-software domains; it provides a retrieval mechanism based on exact matches and similarity metrics which can be customized to specific domains; and it provides a mechanism for defining and ensuring certain semantic relations between attribute values.

***Keywords*. S**oftware reuse library, classification system, taxonomy, similarity, retrieval process, specification language.

1. INTRODUCTION

Complex computer programs have placed a growing demand on the talents of software engineers as well as on existing technologies for software development. In order to keep up with the increasing complexity of today's software systems, productivity must be increased and cost reduced in all phases of the software construction process [2]. An important aspect of the projected solution to this growing demand for new software is the development of support technologies to help increase software reuse, that is, the reapplication of knowledge about one system to other similar systems [1,3]. Rather than starting from scratch in new development efforts, the emphasis must be placed on using already available software assets (e.g., processes, documents, components, tools). This approach avoids the duplication of work and lowers the overall development cost associated with the construction of new software applications. One important characteristic common to most approaches to software reuse is that they rely, either explicitly or implicitly, on some kind of software repository or library from where the "basic building blocks" are extracted. The fact that software libraries are such an important aspect of most reuse systems, has made software reuse library systems (i.e., systems for designing, building, using, and maintaining software libraries) a very important research topic in the area of software reuse [7].

Although these classification models provide the basis for a useful software reuse library system, they have significant limitations and, therefore, can only be regarded as a first step towards a more complete system. They all suffer from one or more of the following problems [4]:

Restricted domain. Some reuse library systems have been designed with the purpose of improving reuse at code level. Their representation language usually does not have the expressive power to model more abstract or complex software domain (e.g. software project, defect, or processes).

Poor retrieval mechanism. One essential characteristic of any software reuse library system is to allow the retrieval of candidate reuse components based on partial or incorrect specifications. This functionality requires the ability to perform similarity-based comparisons, but most systems only provide retrieval based on partial keyword matches or predefined hierarchical structures.

Not flexible. Software reuse library systems must evolve as the level of expertise in an organization evolves. Because of this, a software reuse library system must be flexible enough to allow the incorporation of new classification schemes or new

 ISBN 3-936150-25-7

retrieval patterns, yet this is not the case in most systems.

No consistency verification. Most software reuse library systems are based on representation models, which must satisfy certain basic predicates for the library to be in a consistent state. Yet, most of these systems do not provide a mechanism for ensuring this consistency.

This paper proposes a classification system for software reuse called the Reuse Description Formalism (RDF) which addresses the limitations of current software reuse library systems. RDF is based on the principles of faceted classification, which have proven to be an effective mechanism for creating such systems [8,11]. RDF is capable of representing a rich variety of software (and no-software) domains; provides a powerful and flexible similarity-based retrieval mechanism; and provides facilities for ensuring the consistency of the libraries.

2. FOUNDATION OF RDF

The Reuse Description Formalism uses a generalization of the faceted classification approach proposed by Diaz [10] to represent and classify software objects. The faceted index approach relies on a predefined set of facets defined by experts. Facets and associated sets of terms form a classification scheme for describing components. Component descriptions can be viewed as a records with a fixed number of fields (facets), where each field have a value selected among a finite set of values (terms). Faceted classification scheme has proven to be an effective technique to create libraries of reusable software components. Yet, it suffers from various shortcomings, which limit its usefulness and applicability. The RDF approach to classification overcomes these limitations by extending the representation model as follow [5]:
Components are replaced by instances that belong to several different classes. Instances and classes are defined in terms of attributes and other classes, supporting multiple inheritance.
Facets are replaced by typed attributes. Possible types are: integers, string, enumerations, classes, and sets of the above. Having instances as attribute values allows a library designer to create relations among different instances (e.g., that **push** is a component of **stack**).
The concept of similarity is extended to account for the richer type system, including comparisons of instances of different classes and comparisons of set values.
Semantic attribute relations can be defined and checked using the **assertion** construct. This facility simplifies the process of maintaining the consistency of the definitions in a software library.

An integrated language describes attributes, terms, classes, instances, distances, and their dependencies. Descriptions are type checked. The language is based on a formal mathematical model, which makes it both coherent and analyzable.

2.1. Representation model

To understand the representation principles of RDF, it is useful to consider descriptions of objects of a particular class as point in a multidimensional space, were each dimension is represented by an attribute. Attributes have a name and a list of possible values defined by their associated type (i.e., set of values). If a is an attribute name, and v belongs to the a's type, the assignment "a = v" represent the set all objects whose attribute a is v. Assignment can be combined in expression to define other sets of objects. In particular, if A_1 and A_2 are two assignments, the expression «A1 & A2» represents the intersection of the sets A_1 and A_2. Similarly, "A_1| A_2" represents the union of these sets. In addition, the set of objects that have been defined in terms of a particular attribute a, independently of the value associated with a, is denoted by "**has** a". The set of objects defined by the "has" operator is a short form of the expression "$a = v_1 \mid a = v_2 \mid \ldots \mid a = v_n$" where the value v_i are the elements of the type of a. A set of objects is called a class in RDF. Classes can be given a name they are denoted as **class** (E) where E is an expression; i.e., unions and intersections of other sets of objects. If c is a class name, the set of objects it represents is denoted by "**in** c", and can be combined with other sets of objects in an expression. An object description is called an instance in RDF. Instances can be given a name and they are denoted as **instance** (E) or [E] where E is an expression. Semantically, an instance must have only one set of attributes, therefore we say that instance (E) is well defined if and only if: (1)E is not a contradiction (i.e. , class(E) $\neq \varnothing$), (2) E defines a mapping from attributes to values, that is, E can be simplified into a consistent conjunct of assignments.

Expressions can also be used to characterize particular sets of instances defined in a RDF library. We denote by **set** (E) the set {i| i $\in$ D $\cap$ **class** (E)}, where D is the set of instances in the library. In other words, the **set** operator defines the set of instances in the library that belong to the class defined by E.

2.2. Similarity Model

The goal of any Reuse Library System is to facilitate
the process of finding suitable objects for reuse. RDF supports two criteria for selection candidate

objects: by exact match and by similarity. For exact matches the construct set (E) already described is used. Similarity-based queries are performed using the construct "query E", which denoted the list of instances in the library sorted by decreasing similarity to the target object define by E. That is, the first element of the "query E" is the best reuse candidate for [E], the following element the second best, and so on.

As mentioned earlier, similarity is quantified by a non-negative magnitude called similarity distance, which is used as an estimator of the amount of effort required to transform one object into another. Because of this, distances between two object descriptions, A and B, are not symmetric, because the effort to transform A into B is not necessarily the same as the one required to transform B into A. For this reason, whenever a distance is computed, it is important to define which object is the source and which the target.

Let Z be an object class defined by the set of attributes $Z' = \{A_1,\ldots, A_n\}$, and S and T be two instances in this class. Also, let $S' \subseteq Z'$ be the actual set of attributes used to define S, and similarly for T'. The distance from S to T is denoted by D (S,T) and is computed as follows:

$$D(S,T) = \sum_{A \in S' \cap T'} K_A T_A(S.A, T.A) + \sum_{A \in S'-T'} K_A R_A(S.A) + \sum_{A \in T'-S'} K_A C_A(T.A)$$

Where I.A denotes the value of an attribute A on an instance I. The set $S' \cap T'$ represents the attributes shared by S and T, while S'-T' is the set of attributes found in S but not in T, and similarly for T'-S'. These three sets are disjoint. In addition, each constant K_A is called the relevance factor of attribute A. Their values fall in the range 0 to 1., and must satisfy the relation $\sum_{A \in Z'} K_A = 1$. Functions T_A, R_A and C_A are called comparators, and are explained later in this section.

The expression for distance D(S,T) is based on the assumption that the overall transformation effort from S to T can be computed using a linear combination of the differences between their respective attributes. In other words, attributes are considered independent of each other when computing similarity. This is a strong assumption that limits the types of domains that can be handled by RDF's similarity model.

Relevance factors. In general, the distance between two RDF objects is given by the sum of the distances between their corresponding attributes. This default scheme gives equal importance to all attributes. In our particular situation, this is not a reasonable assumption. For example, one would consider that the difference between component subsystems is more important than the difference between their number of lines of source code. Therefore, the first step required to design comparators is to assign a relevance factor to each attribute in the representation model, that is, to define the amount of influence they have in the computation of similarity distances.

Comparators. Explained earlier, each attribute has three associated functions T_A, R_A and C_A called comparators. T_A is the transformation comparator and is used to qualify the amount of effort required to transform one value of attribute into another. R_A is the removal comparator and is used to estimate the amount of effort required to eliminate a source attribute value not required in the target specification. Finally, C_A is the construction comparator and estimates the amount of effort required to supply a target value not specified in the source specification. The set of all attribute comparators plus their associated relevance factors define a specific similarity model for reuse library. These functions and values must be specified using a process called domain analysis [9] which, among other thing, defines the criteria for similarity for objects in a particular domain. Nonetheless, RDF provides default comparators for each type of attribute. These default comparators can be used as a starting point from which to refine the similarity model of a library. This refinement is normally done by assigning attributes non-default comparators using "foreign" functions specified in some conventional programming language.
RDF defines default comparators for each different kind of RDF type. Although default comparators are well suited for certain domains, sometimes it is necessary to define alternative comparators to be able to capture the semantics and relations of specific objects and attributes. For this purpose, RDF allows the library designer to define arbitrary comparators, which can be assigned to any attribute or type using the "**distance**" clause.

2.3. RDF Specification language

This section presents a formal definition of the syntax of the RDF language. Syntax is presented in a variation of the BNF using the following conventions: Keywords and symbols occurring literally are written in bold; non-terminals are written in italics; type-name, attribute-name, instance-name, term, and class-name all denote identifiers; symbol, … means one or more occurrences of symbol, separated by commas; and $keyword_{opt}$ means that the keyword may or may not occur, without affecting the semantics.

Declarations: A RDF library consists of a sequence of declarations. Each declaration either defines a

name (of a type, an attribute, an instance, or a class) or describes an assertion that must be true of all instances in the library.

Library ::= declaration
Declaration ::= type-declaration | attribute-declaration | instance-declaration | class-declaration | assertion

Attributes and types: Software components and other objects are described in terms of their attributes. We can think of attributes as fields of a record describing the object. The declaration of an attribute specifies the type of the values for the attribute. RDF supports the following types: number, string, term enumerations, object classes, and homogeneous sets of the above.

Attribute-declaration ::= **attribute** attribute-name **:** type**;**
Type-declaration ::= **type** type-name **=** type**;**
Type ::= simple-type distance-clause | **set** distance-clause **of** type
Simple-type ::= number | string | **{**term, ...**}** | class | type-name
Distance-clause ::= **distance**$_{opt}$ | **no distance** | **distance {**triplet,...**}** | **distance *{**triplet,...**}**
triplet ::= term$_{opt}$ → term$_{opt}$ **:** number-literal

The keyword **distance** by itself is optional and assigns default distance functions. The case "**no distance**" indicates that the distance between values of the associated type is always zero. In the third and the fourth forms of the distance clause, the triplet t1 → t2: n means that the distance from t1 to term t2 is n. if t1 is omitted the unspecified value is assumed (i.e., n is creation distance of t2). If both t1 and the arrow are omitted, the previous t1 is assumed. If the keyword distance is followed by the character "*****", then the distances between terms not mentioned in a triplet will be set to infinity. If "*****" is not specified, distances between all terms will be adjusted by computing the shortest path between them.

Expressions: Expression are formed from attribute assignments, the unary operators has and in, and the binary operators **&** (intersection) and | (union).

Expression ::= attribute-name **=** value | **has** attribute-name | **in** class-name | expression **&** expression | expression | expression | **(**expression**)**

The expression "attribute-name = value" means that the value of attribute-name for the instance being defined is value. The expression "**in** class" means that the instance defined belongs to the class; it is similar to a macro-expansion of the expression that defines the class. The expression " **has** attribute-name" denotes the condition that the instance being defined has some value for attribute-name.

Values: Values are used in assignment expressions. Values are either simple values or set values. A simple value is either a literal (number or string), a term, an instance, or the value of an attribute of an instance. Set values must denote homogenous sets; they are described either by extension or by intention, using the

set construct. Only sets of instances can be described by intention.

Value ::= simple-value | **{**simple-value, ...**}** | **set (**expression**)** | **set (**instance-name | expression**)**
Simple-value ::= number | string | term | instance | **self**
| Instance**.**attribute-name | **self.**attribute-name

The construct **set** (E) represents the set of all instances in the library that satisfy the expression (i.e., that belong to **class** (E)). If the optional instance-name is used, the name is bound within E to each instance in the library. The dot notation "instance**.**attribute-name" is used to refer to the value of the attribute attribute-name of an instance. This notation is similar to that used in other languages to access record fields. The keyword **self** is a reference to the instance defined by the expression in which the value is used. Within an **instance** construct, **self** is bound the instance defined. Within an **assertion**, **self** is bound to every instance in the library in turn. Within nested **instance** construct, **self** is bound to the innermost instance.

Classes: A class is defined by giving the corresponding expression; the class denotes the set of all objects for which the expression holds. Classes are used to abstract proprieties of instances and also as abbreviations for the corresponding expressions. Classes are also used as types of attributes whose values are instances.

Class-declaration ::= class-name **=** class**;**
Class ::= **class (**expression**)** | class-name

Instances: Instances are defined in terms of an expression. An instance defined by an expression **E** is a representative of the class of instances defined by "**class (E)**"

Instance-declaration ::= instance-name **=** instance**;**
Instance ::= **instance (**expression**)** | **[**expression**]**

An instance may not exist either because the class is empty (i.e., the expression is a contradiction) or because the class is not specific enough (i.e., it defines more than one valid set of attributes) a sketch of a possible simplification and verification algorithm is as follows.

Expand all "**in**" propositions with the expressions of the corresponding classes.
Transform the expression into disjunctive normal form, as follows:

Restructure the expression using associativity laws so that no disjunction occurs within a conjunction.
Represent each conjunct as a set of assignments and has propositions.

Represent the expression as a set of these conjuncts.
For each conjunction do the following:
Delete redundant assignments.
If there are still two assignment to the same attribute, or there are unsatisfied has propositions, delete the conjunction.

Else, delete has propositions (not needed anymore).
Delete conjunctions that imply another conjunction.
If there no conjunctions left, fail (E is a contradiction)
If there are more than one conjunction left, fail (E is not specific enough)

Assertion: An assertion specifies a semantic constraint that must be true of all instances in the library. Expressions are used to represent dependencies between attributes, to constrain data types and classes, and to enforce correct typing.
Assertion ::= **assertion** expression ⇒ expression;

The meaning of "**assertion** $E_1 \Rightarrow E_2$" is similar to **set** $(E_1) \subseteq$ **set** (E_2). This definition does not capture subtleties with respect to the binding of **self**. RDF signals false assertions

Queries and distance computations: Queries are used to examine a RDF library; they are not part of the library itself. A query command computes a list of instances in the library sorted by decreasing similarity (increasing distance) to the implicit target instance define by an expression. The syntax of queries is:
Query ::= **query** expression | **query** expression **:** identifier
If specified, identifier must be the name of an attribute or a type, and distances are computed using the distance functions associated with the type or the attribute. If identifier is not specified, distances are computed using the default distance functions provided by RDF. The distance command is used to compute similarity distances between a pair of values. This command is useful for verifying the definition of distance functions and the results they produce.
Distance ::= **distance** source-value$_{opt}$ → target-value$_{opt}$ | **distance** source-value$_{opt}$ → target-value$_{opt}$ **:** identifier
The source -value and target-value must be values of the same type (e.g., instance names). In case of terms, they must belong to the same enumeration. If both names are specified, the command computes their transformation distance. If only the source value is given, its destruction distance is computed. Finally, if only the target is specified, its construction distance is computed. The identifier has the same use as in the case of the query command.

3. CONTRIBUTION OF THIS WORK.

As explain earlier, current software reuse systems based on the faceted index approach to classification suffer from one or more of the following problems: they are applicable to a restricted set of domains; they posses poor retrieval mechanisms; their classification schemes are not extensible; and/or they lack mechanisms for ensuring the consistency of library definitions. The primary contribution of this dissertation is the design and implementation of the Reuse Description Formalism [6], which overcomes these problems.

RDF is applicable to a wide range of software and non-software domains. The RDF specification language is capable of representing not only software components at the code level, but it is also capable of representing more abstract or complex software entities such as projects, defects, or processes. What is more, these software entities can all be made part of one software library and can be arranged in semantic nets using various types of relations such as "is-a", "component-of", and "members-of".

RDF provides an extensible representation scheme. A software reuse library system must be flexible enough to allow representation schemes to evolve as the needs and level of expertise in an organization increases. The RDF specification language provides several alternatives to extend or adjust a taxonomy so as to allow the incorporation of new objects into the library without having to classify all other objects.

RDF has a powerful similarity-based retrieval mechanism. One essential characteristic of any software library system is to allow the retrieval of candidate reuse components based on partial or incorrect specification. RDF provides a retrieval mechanism that selects candidate components based on the degree of similarity of their associated library descriptions. This mechanism is based on an alternative refinement process in which components at different levels of granularity can be retrieved. It also includes facilities that allow a library designer to customize the retrieval process by including domain specific function.
In short, RDF addresses the main limitations of current faceted classification systems by extending their representation model.

SUMMARY AND FUTURE WORKS

The RDF is a general system for creating, using, and maintaining libraries of object descriptions with the purpose of improving reusability in software and non-software organizations. RDF overcomes the limitations of the actual systems by extending their representation model and incorporating a retrieval mechanism based on asymmetric similarity distances. In summary, we have presented a software reuse library system called RDF and show how its representation model overcome the limitations of current reuse library systems based on faceted representations of objects. Although the RDF reuse system has to be an effective reuse tool, its performance and usefulness can be enhanced. Several areas that need more research were identified:

Domain analysis**.** In general, to create a library for software reuse it is necessary to perform a domain analysis, the process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest. Domain analysis is currently done by human expert, but several proposals for formalizing and automating this process have been presented in the literature.

Semi-automatic classification**.** A method is needed to classify components in terms of a given representation model. In a general, this involves analysis of the different parts of a component (e.g., source code, documentation, etc.), and the use of heuristics to extract attributes based on this analysis.

Similarity distances**.** A method is needed to test whether the reuse candidates proposed by the system are truly best ones available in the software library. For example, if we classify a new component A know to be similar to a previously classified component B, we would expect the library system to propose B as a reuse candidate for A. failure to do this could arise due to errors in classification of components A or B, or because of errors in the definition of relevance factors and/or distance comparators.

REFERENCES

K.J. Anderson, R.P. Beck, and T.E. Buonanno. The full computing reviews classification scheme, Computer review, 29 January 1988.

B.H. Barnes and T.B. Bollinbger. Making reuse cost-effective, IEEE Software Engineering, January 1991, 13-24.

T.J. Biggerstaff and A.J. Perlis. Software reusability, Volume I: Concepts and Models, ACM Press Frontier Series. September 1989, 474-476,.

Z. Houhamdi and S. Ghoul. A Reuse Description Formalism, ACS/IEEE international conference on computer systems and applications AICCSA01, Lebanese American University, Beirut, Lebanon. 2001.

Z. Houhamdi. Describing and Reusing Software Experience. The international Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications CSITeA'02. Foz do Iguazu, Brazil,June 2002.

Z. Houhamdi. A Classification Scheme for Software Reuse. SCS/IEEE 2002. The third Middle East Symposium on Simulation and Modelling, MESM'2002, Dubai, Emirate united, September 2002.

Z. Houhamdi. Building and Managing Software Reuse Library. The international Journal of Computing and Informatics Informatica (accepted), 2003.

R. Prieto-Diaz. A Software Classification Scheme, Ph.D. thesis, Department of Information and Computer Science, University of California at Irvine, 1985.

R. Prieto-Diaz. Domain analysis for software reusability, In proceedings of the 11th international Computer Software and applications Conference COMPSA'98. IEEE Computer Society Press, 1987.

R. Prieto-Diaz. Implementing Faceted Classification for software reuse, IEEE Transaction on Software Engineering. 1991, 88-97.

R. Prieto-Diaz and P. Freeman. Classifying software for reusability, IEEE Transaction on Software Engineering, January 1987, 6-16.