CLASSIFYING SOFTWARE FOR REUSABILITY

ZINA HOUHAMDI* , SAID GHOUL**

(*) Computer Science Department,University of Biskra BP 145, Biskra RP, 07000. Algeria
E-mail: z_houhamdi@yahoo.fr
(**) Computer Science Institute, University of Philadelphia, BP 1101, Oman. Jordan
E-mail: ghoul_said@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.

The Reuse Description Formalism (*RDF*) is a generalization of the faceted index approach to classification. It was initially designed as a tool to help increase reusability of software components at the *code level* (e.g. functions or subroutines). The goal of this dissertation is to show that *RDF* can also be used effectively to represent and reuse other types of software knowledge.

KEYWORDS: Reuse library, Taxonomy, Classification, Reuse Description Formalism, CCIS library,

1. INTRODUCTION

Current software reuse systems based on the faceted index approach [11,12] to classification suffer from one or more of the following problems [2]: 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, which overcomes these problems [6]:

- *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" [3].
- *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 [4].
- *RDF* provides a consistency verification mechanism. 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. The *RDF* specification language includes an "assertion" mechanism whose purpose is to help specify and ensure the consistency of the object descriptions contained in a library.

In short, *RDF* addresses the main limitations of current faceted classification systems by extending their representation model.

The remaining of this dissertation presents a detailed definition of the *RDF* system. It introduces the concepts behind *RDF*'s representation and similarity models by developing a sample software reuse library. These concepts were formalized [4].

To create and organize reuse library, an extensive domain analysis must be performed beforehand [10]. This analysis must produce a classification scheme (including attributes and their types) as well as an approximate measure of similarity between objects.

This section develops a small software library to classify operations to manipulate data structures consisting of repeated element (e.g., stacks, trees, hash tables). For representation purposes we start with a trivial library and enhance it as more features of *RDF* are introduced.

2. CREATING TAXONOMY

Booch [1] classifies operations over a data structure in the following three classes, based on how the structure is accessed.

- **Constructors**: operations that alter the data structure.
- **Selectors**: operations that evaluate the data structure.
- **Iterators**: operations that visit all element of the structure.

We can describe this simple classification scheme by defining an attribute called function as follows:

Attribute function : {construct, select, iterate};

Another attribute for classification of operations is execution time as a function of the size of data structure.

Attribute timing: {constant, log, linear, loglinear, quadratic, slow};

Attributes function and timing define a simple classification scheme that can be used to describe four operations for stack manipulation. Each of these descriptions is called *instance*.

Push = [function = constructor & timing = constant];

Pop = [function = constructor & timing = constant];

Top = [function = select & timing = constant];

New = [function = constructor & timing = constant];

This section has introduced two basic concepts of *RDF* language: attributes and instances. The type associated with both attributes is an enumeration of terms. Each instance defines the attribute values of a particular data structure operation.

3. EXTENDING TAXONOMY

The characterization of the functionality of operation presented above is too coarse. In fact, the descriptions of push, pop and new are identical. This section refines this characterization by extending the classification scheme. There are at last three approaches to do this.

- Add or replace terms in the type of attribute.
- Add more attributes.
- Describe attribute values in terms of more primitive attributes.

The first two approaches are common practice while designing a taxonomy and the only alternatives a library designer has with other classification systems such as AIRS or faceted classification system. The third approach is unique to *RDF*, and allows the construction of hierarchical classification system.

3.1 Adding values to a type

In this approach, the classification scheme is refined by including additional values to the type of an attribute. In particular, we add new terms to the functionality attribute. In the context of data structures consisting of repeated elements, the constructor term will be replaced by three new terms create, insert, and remove. With this new definition we can now tell push from pop and tell those from new. The updated definitions are as follows:

Attribute function : {create, insert, remove, select, iterate};

Push = [function = insert & timing = constant];

Pop = [function = remove & timing = constant];

Top = [function = select & timing = constant];

New = [function = create & timing = constant];

This drawback of this approach is that instance definitions had to be manually modified (e.g., changing construct by the corresponding new term in each instance). Moreover, these extensions create flat taxonomies with few attributes and many terms, instead of hierarchies.

3.2 Adding attributes

In *RDF*, it is possible to define a new attribute and then use it to refine the classification of selected instances. Unlike other faceted classification system, this new attribute does not have to be used in all instances. Hence, the addition of attributes requires modifying only those instances for which the new attribute is meaningful and important.

For example, we extend the taxonomy by adding a new attribute called *exception*. This attribute is used to describe those operations that can signal a fatal exception such as a stack overflow or underflow. The following definitions are add or modified in our library:

Attribute exception : {underflow, overflow};

Push = [function = insert & timing = constant & exception = overflow];

Pop = [function = remove & timing = constant & exception = underflow];

Only those operations that can generate an exception (push and pop) have been described using the attribute exception. The remaining in the library (top and new) were not modified and, therefore, have no defined value for the attribute exception.

It can be argued that the attribute exception could have been defined with an additional term called *noexception* to describe those operations that do not generate exceptions. In this solution, all instances would been defined using the same set of attributes and therefore a system like AIRS could still be used to model our taxonomy. Although this argument is valid in the current example, in fact that *RDF* can handle descriptions with different sets of attributes in particularly important in the case of libraries containing objects of different classes such as "project", "systems", "packages", and "operations". The attributes of these

sample classes are most probably disjunct, but they can all be classified in a single library.

3.3 Describing values of an attribute

RDF provides a new approach to extend a classification scheme [5]: describe all terms of an attribute using more primitive attributes. This process is illustrated by refining again the functionality attribute.

Within the domain of data structure consisting of repeated elements, the functionality is described in term of three new attributes: access (whether the data structure is written or only read), target (which elements are affected), and newsize (how the number of elements varies).

```
Attribute access  : {write, read};
Attribute target  : {leftmost, rightmost, keyed, any, all, none};
Attribute newsize : {increase, decrease, reset, same};
```

These new attributes are used to define each of terms that belong to the attribute functionality.

```
create  = [in constructors & newsize = reset & target = none];
insert  = [in constructors & newsize = increase];
remove = [in constructors & newsize = decrease];
select  = [in selectors];
iterate  = [in iterators];
```

Where constructor, selectors, and iterators each define a class of instances. The class mechanism is used both as an abstract mechanism and, also, as an abbreviation for expressions. These classes are defined as follows:

```
Constructors = class (access = write);
Selectors    = class (access = read & newsize = same);
Iterators    = class (target = all);
```

The definition of the attribute functionality can now be changed, because its element no longer belong to enumeration type to a class of instances, namely the class of instances defined in terms of one or more of the attributes access, target, and newsize.

```
Attribute function : class (has access | has target | has newsize);
```

Since all former terms of attribute function are defined, instances described using these values (e.g., push) do not need to be redefined. That is, this extension of the classification system does not affect the classification of objects already in the library.

This extended classification scheme allows us to define new categories of functionality. For example, we can define modify as a possible value of functionality, and also describe more specific iterators.

```
Modify            = [in modifiers];
passive_iterate  = [in iterators & in selectors];
active_iterate    = [in iterators & in constructors];
modify_iterate   = [in iterators & in modifiers];
modifiers        = class (access = write & newsize = same);
```

Where modifiers is the class of all operations that update elements in the data structure.

In summary, the process required to extend a classification scheme by redefining the terms of the attribute is as follows:

1. Select an attribute *a* whose terms era to be refined. Let T be the type of *a*. In the example, *a* = functionality and T = {create, insert, remove, select, iterate}.
2. Perform a domain analysis on the domain of the terms of a. From this analysis, define a set A of new attributes that describe terms in T, and determine the type for each attribute in A. In the example, A = {access, target, newsize} with their corresponding term enumerations.
3. Redefine attribute a. possible values for a are not terms as before (type T is no longer part of the library), but instances that belong to a class defined using the attributes in A.
4. Define each former term $t \in T$ as an instance using the attributes in A, following the same procedure used to describe data structure operations.
5. If needed, other values for a can be described. This values can be specializations of former terms (e.g., passive_iterate) or they can represent new concepts (e.g., modify).

In principle, this process of refinement can be done indefinitely providing deep hierarchical taxonomies, but there is a point in which using this formalism is no longer useful (e.g., do not use *RDF* to describe detailed functionality, including pre- and post-conditions).

4. CREATING OBJECT HIERARCHIES

Reusable software usually consists of packages or modules, made from operations and heir packages. We want to represent this modular structure, but we do not want to force any granularity of reuse. That is, we want to have a library consisting of packages and operations, assuming that both complete packages and isolated operations will be reused. The following declarations define the kinds of reusable software components for a library of data structure packages. Because a package can have several subunits, the subunits attribute has a set type.

```
Attribute subunits : set of components;
Attribute parent   : packages;
Components = class (in packages | in operations);
Packages     = class (has subunits);
Operations  = class (has function | has timing);
```

Two other attributes for packages are defined: maxsize (whether there are limits in the number of elements of the structure) and control (whether concurrent access is

supported).

Attribute maxsize : {bounded, limited, unbounded};

Attribute control : {sequential, concurrent};

With these declarations, a stack package comprising the operations already described can be defined using one extra attribute (parent). The implementation has no preset bound on size and does not provide support for concurrency.

Stack = [subunit = set (parent = stack) & maxsize = unbounded & control = sequential];

Push = [parent = stack & function = insert & timing = constant & exception = overflow];

Pop = [parent = stack & function = remove & timing = constant & exception = underflow];

Top = [parent = stack & function = select & timing = constant];

New = [parent = stack & function = create & timing = constant];

Where the construct "set (parent = stack) denotes the set of all instances defined in the library for which the attribute parent is equal to stack, in other words, the set {pop, push, top, new}.

5. DEPENDENCIES AMONG ATTRIBUTES

All classification schemes assume that certain semantic relations between attributes values are being maintained. For this purpose, *RDF* provides a mechanism that uses assertions to define semantic constrains between attribute values.

For example, consider the case of attributes describing the functionality of an operation. If the data structure is not written then there is no size change, and if the structure is reset then there is no specific target. These two relations can be expressed as follows:

Assertion access = read ⇒ newsize = same;

Assertion newsize = reset ⇒ target = none;

In addition, the attribute maxsize and control are only relevant for packages, and all units that declare a package as their parent must indeed be subunits of the package.

Assertion has maxsize | has control ⇒ in package;

Assertion in packages ⇒ subunits (parent = self);

The keyword ***self*** denotes the instance being analyzed for compliance with the assertion.

6. DEFINING SYNONYMS

One of the difficulties of describing operations given our current taxonomy is remembering the precise terms used in the library. Besides, certain concepts can be given or referenced by more than one name. The introduction of synonyms for terms has been suggested as a partial solution to this problem.

One could declare that distance between two terms is zero, making them synonyms from the point of view of queries based on similarity. However, queries based on exact matches will considered them different. In *RDF* is possible to declare an identifier i_1 to be a synonym of an identifier i_2 by simply declaring $i_1 = i_2$. For example:

Update = write;

Preserve = read;

These definitions introduce the synonyms update and preserve for the terms write and read of attribute access, respectively.

7. QUERIES AND COMPARING OBJECTS

In order to find reusable software components in the library of packages and operations; it is necessary to define the distance values associated with the terms of enumerations types. This allows *RDF* to compute distances not only between these terms, but between instances defined using these terms.

Distances between terms are defined with a distance clause. For example attribute access and newsize and their distance clauses are given below. The distances shows here are just sample values. {the process of assigning distances is not described in this paper because the emphasis is not on how to define similarity distances between object}.

Attribute access : {write, read}

distance {write → read: 4 , read → write: 6};

Attribute newsize : {increase, decrease, reset, same}

distance {increase → decrease: 5, same: 7, decrease → increase: 5, reset: 3, reset → same: 10, same → reset: 10};

By transitivity, we can determine other distance not explicitly given. For example, the distance from increase to reset is 5 + 3 = 8, and the distance from decrease to same is 12. Note that a bigger value for this distance (13) can be obtain going from decrease to reset to same, but *RDF* always uses the smallest value.

Basically, the distance between two instances is computed by adding the distances of their corresponding attribute values. For example, the distance from remove to select is 16, given by the distance from write to read (4) plus the distance from decrease to same (12).

Remove = [access = write & newsize = decrease];

16 = 4 + 12

select = [access = read & newsize = same];

Distances between instances are used by *RDF* to select reuse candidates from a library. This selection is performed using the query command. For example, the following query finds components that are similar to an operation that retrieves an arbitrary element from a data structure in at most logarithmic time.

```
Query function = [in selectors & target = any] & timing = log;
```

Consider another example. Find a data structure with three operations : one to initialize, one to insert an element, and one to traverse the structure without modifying it; concurrent control is not needed, but the structure must be able to handle an unbounded number of elements.

```
Query maxsize = unbounded & control = sequential & bunits =
      {[function = create], [function = insert], [function =
      passive_iterate]};
```

In this query, only the functionality of the operations have been specified. Attribute timing is not defined; meaning that any value for timing is equally acceptable in the retrieved operations.

8. SAMPLE RDF TAXONOMY

RDF was initially designed as a tool to help increase reusability of software components at the *code level* (e.g. functions or subroutines). The goal of this section is to show that *RDF* can also be used effectively to represent and reuse other types of software knowledge.

In this section we describe a taxonomy for classifying the different modules and functions that compose the CTC CCIS library and creating a *RDF* reuse library with the purpose of facilitating their reuse. The CCIS library developed at Contel Technology Center (CTC) is composed of several modules implemented in C [7]. These modules are used to implement the basic functionalities of Command, Control, and Information Systems.

- *General* (GEN): general purpose functions that do not belong to any specific module. These functions are typically extensions to the ones contained in the standard C library.
- *Memory file* (MF): implements sequential files allocated in main memory (RAM). These files are created and exist only during the execution of a program.
- *Set Str*ucture (SET): implements unbounded sets of elements. The elements of a particular set must be of the same type.
- *Database Interface* (IDB): provides a simplified interface to the most commonly used operations of a relational database system.
- *Database File* (DBF): implements a specialized form of database files. These files are flat structures stored in a relational database processor.
- *Mail Service* (MS): implements the basic functionalities of an electronic mail system.
- *Man-Machine Int*erface (MMI): implements a graphic user interface based on windows, predefined keys, and menus.
- *Free Text File* (FTF): implements a specialized form of text files which are stored in and retrieved from on a relational database.
- *Parametric Database Display* (PDD): collection of parametric functions used to retrieve and display information contained in a relational database.

As with *RDF* GRACE library, the *RDF* CCIS library included two types of objects: *modules* and *functions*. The former represent the different C modules of the CTC CCIS library, and the latter represent their associated C functions. Modules are described using four attributes according to the following class definition:

```
Module = class (has mdAllocation & has mdIterator & has mdService &
                has mdOpers);

Attribute pkName    : string;

Attribute pkIterator     : {Iterator, nonIterator};

Attribute pkAllocation : {Bounded, unBounded, Limited};

Attribute pkOpers       : set of Operation;

Operation = class (has opType & has opKey & has opCount & has
                   opTarget & has opRange & has opDirection & has
                   opPackage);

Attribute opType : {Create, Select, insert, Remove, Traverse, Query};
```

The attribute opTarget indicates the type of the data structure elements affected or selected by the operation. This may be either a set of nodes, one node, or a link between nodes. The number of elements affected or selected is defined by the attribute opCount, and the attribute opKey indicates the type of key value used to select elements in the structure.

```
Attribute opTarget : {Nodeset, Element, Link};

Attribute opCount  : {All, One, Zero};

Attribute opKey     : {Index, Pointer, Value, Size};
```

The attribute opRange and opDirection are used to define the relative location of the elements affected or selected by the operation. The former indicates a range of elements within the structure. The latter, defines a direction, relative to the value of opRange, on which the component will operate.

```
Attribute opRange       : {Firstlast, Firstto, Fromlast, Fromto, Rest,
                           Floating, First, Second, Last};

Attribute opDirection : {Left, Right, toright, toLeft, Breadth, Depth};
```

Finally, the attribute opPackage defines the package to which the operation belongs. It is defined as follows:

```
Attribute opPackage : Package;

Package = class (has pkName & has pkAllocation & has pkIterator &
                 has pkOpers);
```

The attribute mdService describes the services provided by the functions of the package (e.g., memory management, mail delivery, etc.). The definition of this attribute is given below.

```
Attribute mdService : {GEN, SET, MF, IDB, MS, DBF, FTF, MMI,
                       PDD};
```

The functions of each package in the *RDF* CCIS library

were described in terms of two attributes: fnFunction and fnObject. The fnFunction attribute describes the functionality of a component, and it is defined as follows. These terms were extracted from the documentation of the CTC CCIS library [7].

Attribute fnFunction : {add, assign, clear, close, convert, copy, count, create, delete, display, enable, execute, find, goto, intersect, log, map, measure, modify, open, parse, process, read, rename, replace, retrieve, search, suspend, terminate, test, transfer, union, write};

The attribute fnObject describes the kind of object produced or consumed by the function, and is defined as follows:

Attribute fnObject : {address, code, column, column_type, control_variable, descriptor, directory, element, event, file, function-key, group, interface, keyboard, list, menu, name, offset, owner, pdd_descriptor, pdd_page, permission, pointer, pdd_table, printer, queue, subset, queue_entry, record, set, sql_command, string, substring, text, tuple};

One of the difficulties of posing queries in a library so rich in terminology is remembering the precise terms used to describe functions. To facilitate this situation, the *RDF* CCIS library included a list of synonym definitions for some of the terms of the attributes fnFunction and fnObject. The following are some sample synonym definitions:

update = write;	sequence = string;
insert = add;	locate = address;
remove = delete;	node = element;

These synonym definitions were made part of the *RDF* CCIS library by including them as terms of their respective attributes, and then defining the distance between them and their synonym terms as zero

9. CONCLUSION

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 [2,3,4].

RDF overcomes part of the limitations of current faceted system by extending the their representation model. Two main concepts form the core of *RDF*'s representation model: instance and classes. Instances are descriptions of reusable objects, while classes represent collections of instances with a set of common properties. Objects are described in terms of attributes and associated values. Unlike faceted classification, which is limited to having only terms as attribute (facet) values, *RDF* allows attributes values to be instances and even sets of instances.

This generalization can be used to create one-to-one, one-to-many, and many-to-many relations between different object classes within a library. In other words, *RDF*'s specification language [4] is powerful enough to represent a wide variety of software (and non-software) domains, ranging from standard software components such as data structure packages and their operations, to more complex domains such as software defects and software process models. In addition, *RDF* language provides facilities for ensuring the consistency of the libraries.

Yet, no evaluation has been performed on RDF's similarity-based retrieval mechanism. Towards this end, we are currently developing a reuse software library– based on information contained in the Software Engineering Laboratory (SEL) database [8]. This database contains thousands of records containing functional and structural descriptions, a well as statistical data, related to hundreds of projects developed at the NASA Goddard Space Flight Center. In addition, this database contains information regarding the origin of the project components [9], which indicates whether they were implemented from scratch or by reusing other components at NASA. This reuse history will allow us to evaluate our similarity-based retrieval mechanism by comparing the reuse candidates it proposes with the ones that were actually used at NASA.

REFERENCE

[1] G. Booch, "Software Components with Ada", *Benjamin-Cumming Publishing Company*, Menlo Park, California, 1997.

[2] Z. Houhamdi and S. Ghoul. "A Reuse Description Formalism". *ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'2001*, Lebanese American University, Beirut, Lebanon. 2001.

[3] Z. Houhamdi and S. Ghoul. "A Classification System for software reuse". *Fifth International Symposium on Programming System, ISPS2001*, USTHB Computer science Institute, Algiers, Algeria, 2001.

[4] Z. Houhamdi. "A Specification language for software reuse". *CSS/IEEE Alexandria Chapter. 11th International Conference On computers: Theory and Application, ICCTA2001*, Head of Electrical Control, Alexandria, Egypt, 2001.

[5] Z. Houhamdi. "Developing a Reuse Library". *CSS/IEEE Alexandria Chapter. 11th International Conference On computers: Theory and Application, ICCTA2001*, Head of Electrical Control, Alexandria, Egypt, 2001.

[6] Z. Houhamdi. "An adaptative approach to reuse software". *SCS/IEEE 2001. The third Middle East Symposium on Simulation and Modeling, MESM'2001*, Amman University, Amman, Jordan, 2001.

[7] Z. Houhamdi. "Software Reuse: a new classification approach". *The International Symposium on Innovation in Information and Communication Technology, ISIICT'2001*, Philadelphia University, Amman, Jordan, 2001

[8] R. Kester. "SEL Ada reuse analysis and

representation". *Technical Report, NASA Space Flight Center*, Greenbelt, Maryland, November 1990.

[9] "NASA Goddard Space Flight Center", Greenbelt, Maryland. *Software engineering Laboratory (SEL*) database Organization and User's guide, revision 1 edition, February 1999 (internet).

[10] R. Prieto-Diaz, "Domain analysis for software reusability", *In proceedings of the 11th international Computer Software and applications Conference (COMPSA988).* IEEE Computer Society Press, October 1998, pp. 23-29.

[11] R. Prieto-Diaz and G. Jones, "Building and managing software libraries", *In proceedings of the 12th international Computer Software and applications Conference (COMPSA'97*), IEEE Computer Society Press, Chicago, October 1997, pp. 228-236

[12] R. Prieto-Diaz, "Implementing faceted classification for software reuse", *Communication of the ACM*, 2000, pp. 88-97.