International Journal of Software Engineering and Knowledge Engineering
Vol. 3, No. 3 (1993) 287–318
© World Scientific Publishing Company

ADA REUSABILITY AND SYSTEM DESIGN ASSESSMENT USING THE DATA BINDING TOOL*

ALEX DELIS and VICTOR R. BASILI†
Department of Computer Science, University of Maryland, College Park, MD 20742
{ad,basili}@cs.umd.edu

Received 20 April 1992
Revised 10 March 1993
Accepted 29 March 1993

This paper reports on the development of the data binding tool and its use in Ada source code reusability and software system design assessment. The tool was built around the metric of data bindings. Data bindings fall in the category of data visibility metrics and are used to measure inter–component interactions. Software system components are defined in the context of the Ada language using a flexible scheme. They are used, along with cluster analysis, to present structural configurations of a software system. The clustering technique as well as the tool design and its problems are discussed. The analysis of dendrograms (trees of components produced by the tool) reveals several classes of systems dendrograms and provides a simple mechanism for Ada source code reusability. Finally, the implications of different design methodologies used to develop the test software are discussed and explanations for the several types of dendrogram formulations are given.

Keywords: Data bindings, system structure, reusability, design assessment.

1. Introduction

Software projects are often scaled down, delayed and even abandoned due to cost and time constraints set on the development effort. Although we have experienced a great number of innovations like the introduction of workstations and programming environments, the development of software has not achieved a sufficient rate of productivity to satisfy demand. It is expected that this trend will continue, at least in the near future [10, 35]. There are two major reasons for this: limited software reuse is employed [6, 7] and assessment of software designs is often hard to be carried out [21, 14, 28].

By reuse, we mean not only reuse of code, but also of other life–cycle products as well as processes. In certain environments, the reuse of processes is already a

*This work was supported in part by the US Army Institute for Research in Management Information and Computer Science (AIRMICS) under Grant AIRMICS–01–4–33267.
†Also with the University of Maryland Institute for Advanced Computer Studies (UMIACS).

common practice. People follow a predefined path in their problem solving. Based on previous project experiences, personnel and resources are allocated and schedules and milestones are laid out. However, reuse of products still remains limited. Although reuse of early software life cycle products could prove extremely profitable, reuse of source code itself remains an issue of critical importance [10].

Design represents an intermediate, yet extremely essential, part of any project [27, 21]. High level solutions are offered to a set of problems furnished by the requirements analysis phase. Programmers are expected to comply with the outlined design plans on their way to implementation. In an environment in which many people work simultaneously to meet development goals, and in which changes of design are common, regular inspection and validation of the design blueprints is desired. Compatibility of the design with the released code should be observed as well.

The introduction of Ada [30] had two primary goals: to promote reusability of code through the wealth of the language constructs and to assist system design through its abstraction mechanisms. The elementary building block of Ada is the subprogram structure [9]. Ada offers packages, tasks and generics as major structural elements. Packages provide the capability to extend the language by creating new objects with their operations. Tasks provide concurrent interaction among language objects. Generics offer a versatile mechanism for building reusable software components. There is a great deal of interest in using Ada as a design language, given the benefit of design analysis provided by the numerous tools implemented for the analysis of Ada source code [32]. While it will not expected that full elaboration of the source code will be available at design time, the system architecture and the interfaces between system components will be identified. Analysis of the architecture and inter-module information flows provides indicators of product qualities such as reliability, maintainability and reusability.

The purpose of this paper is to report on the development of an Ada based tool (the Data Binding Tool or called **dbt**) and to demonstrate how it assists in the reuse of source code and system design assessment of Ada systems. The tool has been developed around the idea of information flow among software components. Data bindings [4] provide a measure of component and module interaction. They evaluate the proximity of the system's components. Those "closeness" measures serve as input to a mathematical taxonomy (cluster analysis) method that constructs a simple and unique tree diagram of the elements involved. This diagram — called a dendrogram — expresses element similarities and dissimilarities at a glance. By examining dendrograms, one can determine what happens when certain components within a cluster need to migrate into a new environment. Designers can determine if clusters are representative of their design and can derive a deeper understanding of their systems. The tool has been applied to a number of software projects and has helped to draw conclusions about reuse potentials. The analysis, from the point of view of design assessment, shows that indicators of quality factors vary according to the employed design techniques. Conformance of the obtained dendrograms with the initial designs of the systems is also discussed.

The organization of the paper is as follows. In Sec. 2, a review of related work and the concept of data bindings are presented. The notions of a *component* and data bindings for Ada are discussed in Sec. 3. Section 4 briefly studies the concept of mathematical taxonomy and how the final output (a dendrogram) is produced. In Sec. 5, the design of the tool and the major problems encountered are examined. Section 6 presents issues in reusability and design methods used, sets questions to be answered through the dendrograms, gives a description of the test data and offers an explanation of the derived dendrograms. The results of our experiments are discussed in detail. Conclusions and research plans can be found in the last section of the paper.

2. Background

2.1. *Related work*

Information flow metrics and concepts — similar to the one used in this paper — have been extensively used in the literature for a broad range of purposes. Their benefits are that 1) they can be used in stages prior to detailed coding (information flow metrics can be applied during PDL design), 2) they provide insight into complex programming structures, and 3) they can be readily automated since a simple instrumentation of the compiler is usually required. In this section, we present only a partial overview of previous research in the area from a large body of literature.

In [18], the concept of information flow is formally defined and the notions of fan–in and fan–out are introduced. Global flows are distinguished from local flows. Based on these concepts the authors speculate that the complexity of a procedure depends on the complexity of its code and its environment. Measurements for the suggested metrics were used to locate design and code problems. Information flow was used to quantify the strength of connections between program modules. Wilson and Osterweil [37] used information flow to detect mistakes in C programs. Code that goes through the compilation phase successfully may still have problems at run–time. The key idea is that variables should follow a sequence of events: definition, reference, undefinition. If a variable of a function presents a pattern such as definition–definition, definition–undefinition, or undefinition–reference somewhere in the control flow, then it is a data flow anomaly.

Barth [1] used data flow techniques to perform inter–procedural flow analysis. The goal of this analysis was to determine information available at particular program segments. Segment semantics is propagated through the program in a way that reflects the control structure. Generally, the problem of global flow analysis is shown to be NP–hard. A one–pass algorithm is used to gather complete inter–procedural information. The emphasis of this algorithm is on the computation of modification and usage information for every object (similar to the notion used in data bindings, and in [37]). Information flow techniques were used in [36] by Stevens to demonstrate improvement in application development productivity. Indeed, this is the first reference which views information flow techniques in conjunction with reusability and system design assessment, regarding them to be a mechanism for reducing system complexity. Belady and Evangelisti [5] used the interconnection of program modules and data structures, in terms of calls and references, to determine system partitioning through the use of clustering techniques. An algorithm to perform automatic clustering of modules and a metric to qualify the complexity of the resulting module partitioning were proposed.

Hutchens and Basili [19] presented an evaluation of system component interaction in Fortran using data bindings. Bindings among system subprograms were input to a number of clustering algorithms to derive system dendrograms. The purpose of that work was to find functional clusters, to perform error analysis involving changes in the code and finally to compare some clustering techniques. Selby and Basili [33] used data bindings to quantify ratios of coupling and cohesion. They subsequently used these ratios to generate hierarchical systems descriptions in order to localize errors by identifying error–prone system structures during the development phase.

2.2. *Data bindings*

Data Bindings fall in the category of measures for data visibility [4]. They have been utilized to measure the interaction among system segments (a segment is a set of executable statements and conceptually is very similar to the notions of module and component). The definition of data bindings follows.

Definition:

Let α and β be two program segments and variable γ be global to both. If γ is assigned a value by segment α which is accessed (or referenced) by β, then there exists a data binding between the two program segments, denoted by the triplet (α, γ, β).

This triplet conceptually describes a flow of information from the first segment to the second. It is also possible that another binding indicating a reverse flow exists (i.e., (β, γ', α) where γ' is a global variable assigned a value by β and referenced (accessed) by α). Intra–segment bindings are not considered to be of interest since they portray flow internal to the segment (so (α, x, α) does not count). Although the definition speaks about *assignments* and *accesses*, this really refers to *potential* assignments and references.

In [19], the notion of segment was made to be synonymous to the Fortran subroutine or function. Several families of Data Bindings were identified in the same early work [19]: potential, used, actual and control flow. The definition given above is for *actual data bindings*. It is also the only one to be used in realistic measurement settings [4, 19, 33]. Potential bindings, as the name suggests, capture the *possibility* that two segments communicate through a variable located in both their lexical scopes. So, if α and β are two segments and γ is in the lexical scope of α and

β then (α, γ, β) is a potential data binding. Used bindings reflect the similarity of two segments with regard to the "use" (either reference or assignment) of a variable in their scope. Thus, if α and β are two segments that use global variable γ, then there is a used data binding (α, γ, β). Finally, control bindings are an improvement over actuals in the sense that an extra condition is required, namely control from segment α is passed over to segment β. Naturally, the number of potential bindings is the largest of all binding types. As the definitions become more restrictive, smaller numbers of data bindings are found.

3. Concepts in Ada

3.1. *Segments*

System segments in [19] are called either modules or components. The term *module* is unfortunately overloaded in the literature. There is disagreement on what should be considered as a module (component). Myers [29] suggests that a system module (component) is a set of executable statements satisfying the following criteria:

- It is a closed subroutine.
- It has the potential of being called from any other module in the program.
- It has the potential of being independently compiled.

The last two requirements are more suggestive than definitional. Fortran subprograms generally comply with all of the above criteria. Since subprograms are the only constructs for abstraction in Fortran, their utilization as system components in [19] is justified.

Hammons in [16] defines Ada modules as non-nested subprograms. However, subprograms encapsulated in package bodies are not characterized as modules. The claim is that since such subprograms cannot be called from any random point in the system (only within the package body's scope) they do not qualify as modules. Hammons also suggests that Ada tasks do not qualify either.

Ada provides a wealth of programming constructs and it is generally difficult to identify one of them as the general modularization mechanism. Packages mainly accommodate the need for encapsulation and abstraction. The main routine that drives an Ada system is a subprogram. Therefore, it is difficult to differentiate between packages and subprograms. A flexible scheme, called Ada data Binding Components (ABCs hereafter), is proposed here to define Ada constructs as components. ABCs offer a two-level module definition capability.

At the first level, subprograms (functions and procedures) as well as task entry bodies constitute the system components. There is not much syntactic difference between a task entry call and a procedure call. Although the nature of tasking is dynamic, information flow among task entries can be modeled and analyzed from a static perspective. Indeed, the data binding concept could be applied to the entries. Entries are considered to behave much like procedures. Entry parameter lists correspond to formal parameter lists. In addition, bodies of tasks can be compiled separately. ABCs of the first level are the essential building blocks of the language and of our analysis. Note that these low-level components comply with Myers' laws.

At the second level, certain sets of components of the first level are viewed as integrated entities. A substantially different view of a system under analysis could be taken by recognizing that there is such a need for integration. That would change the formulation of the participating ABCs in the analysis. For instance, there are occasions where packages implement abstract data types and they must be seen as integrated components. The same applies to the case of nested functions, subprograms as well as tasking constructs encapsulated either in subprograms or packages. Thus, a mix of higher-level components (packages) with elementary ones (procedures, functions) may be obtained, providing a more diversified view of the system.

The ability to express two-level ABCs is supported by our work. If nothing is explicitly demanded, then the analysis will be carried out assuming that only first level components are elaborated. Otherwise, the second level ABCs will be built on knowledge acquired by the analysis performed on the first level of Ada data Binding Components.

Block statements are not considered ABCs of either level. Blocks are instead part of the Ada Binding Components that contain them within their scope. Package body initializations are also not ABCs. Initializations are considered to be part of the package as a whole, therefore, they are covered by the Ada Binding Components of the second level. It is also important to understand and classify the interaction of instantiated generics with the rest of the system. Instantiated ABC generics are identified and their data interaction with other Ada Binding Components is evaluated.

3.2. *Definition of data bindings in Ada*

The definition of Data Bindings, in light of the Ada Binding Component scheme, is modified as follows:

Definition I:

There exists a Data Binding (abc_i,x,abc_j) between two ABCs abc_i, abc_j if all of the following hold:

- abc_i calls abc_j,
- object x is part of the abc_j interface (either an element of the formal parameter list or a returned function value),
- one of the abc_i,abc_j assigns x and the other references it.

Note that in Ada the mode of the formal parameter x determines the direction of the binding. If it is an *IN* parameter type then the binding (abc_i,x,abc_j) is established. If it is an *OUT* then the reverse direction binding is set up (i.e., (abc_j,x,abc_i)).

In the case of *INOUT*, either binding could be established depending on who assigns and who references.

Definition II:

There exists a Data Binding (abc_i,x,abc_j) between two ABCs abc_i, abc_j if:

- The scope of object x extends to both abc_i and abc_j.
- abc_i assigns to x and abc_j references it.

The scope represents the set of system variables that are accessible by both ABCs. Some of these objects may be local to the ABCs library units. They may also be objects belonging to different library units than those of the ABCs, but are visible by being *WITH*ed. Except for the removal of the stipulation that x be a global variable, this last definition parallels the original one [4]. These definitions can be applied whether or not the components in question are visible at the library level or nested inside one another. The selected level of ABC for a particular construct determines exactly what the scope of each segment is. Note also that the defined bindings are those established through execution and not through elaboration (such as through initializations or default value assignments).

A two-dimensional matrix is used as the recording mechanism for all occurrences of data bindings among the ABCs of a system. This data binding matrix is used as input to a mathematical taxonomy that determines the several clusters. This is discussed in the next section.

4. Mathematical Taxonomy

A Mathematical Taxonomy (or Cluster Analysis) is used to group similar objects. The similarity of objects is based on properties of the objects. The role of clustering is multiple. It groups, displays, summarizes, predicts and provides a basis for understanding. Items (or objects) are grouped to create more general and abstract entities that share properties and have identical behavior in the context of the system from which they are derived. Clusters of objects are displayed so that differences and similarities become apparent. Properties of clusters are highlighted by hiding properties of individuals. What is expected in general from a mathematical taxonomy is that clusters present *similar* properties. Thus, clusters easily isolated offer a basis for understanding, and speculations can be derived about the structure of the system. Unusual formulations may reveal anomalies.

In this paper, Mathematical Taxonomy is used as the tool to produce ABCs groups. These groups form the basis for a classification scheme for evaluating the system design and future partial system reuse. A great number of algorithms for Mathematical Taxonomy has been proposed in the literature [23, 15, 17, 20]. Given that programs are often organized as hierarchies of elements and the two-level flexibility that the ABC scheme provides, a bottom-up clustering algorithm appears to be the most appropriate [19, 23].

In general, the initial *raw* data collected on a set of n objects (having m attributes) constitutes a matrix M with size $n \times m$.

$$M = \begin{pmatrix} \mu_{1,1} & \mu_{1,2} & \cdots & & \mu_{1,m} \\ \mu_{2,1} & \mu_{2,2} & \cdots & & \mu_{2,m} \\ \cdots & & & & \cdots \\ \mu_{i,1} & \cdots & \mu_{i,j} & & \\ \cdots & & & & \cdots \\ \mu_{n,1} & \cdots & & \cdots & \mu_{n,m} \end{pmatrix}$$

Element $\mu_{i,j}$ is the score of the i–th object for the j–th characteristic. The first computation of the Cluster Analysis method is to produce a matrix N out of matrix M. Matrix N is called the *distance matrix* (informally, it expresses how far away every object is from all others) and has size $n \times n$. This distance matrix is passed over to an iterative algorithm that constructs the objects' dendrogram [23, 15]. The rest of this section describes in detail the formulation of the initial as well as the distance matrix and discusses the iterative algorithm that is used.

The clustering of Ada data Binding Components matrix M is formulated with the use of data bindings. In this case, $m = n =$ number of ABCs and the properties correspond to the number of data bindings every ABC maintains with all the rest (i.e., $\mu_{i,j}$ is the number of any direction bindings between components i and j). M is a symmetric matrix. The transformation of M into the distance matrix N is performed using the formula:

$$N_{i,j} = \frac{\sum_k \mu_{i,k} + \sum_k \mu_{k,j} - 2\mu_{i,j}}{\sum_k \mu_{i,k} + \sum_k \mu_{k,j} - \mu_{i,j}} * 100$$

The numerator of the expression is the number of bindings in which either the i–th or the j–th component participates but not both. The denominator expresses the number of data bindings in which either the i–th or the j–th component participates, or both. Their fraction $N_{i,j}$ represents the probability that a data binding chosen from the set of bindings that involve either i or j is not in their common set of bindings. If $\mu_{i,j} = 0$ (no bindings occur between the two ABCs) then $N_{i,j} = 100$. This expresses that the dissimilarity (or distance) between i and j is as large as possible. Consequently, the more bindings between any two components the smaller their distance is.

For example given the matrix M with elements (a, b, c, d, e):

$$M = \begin{pmatrix} 0 & 2 & 0 & 2 & 1 \\ 2 & 0 & 1 & 3 & 2 \\ 0 & 1 & 0 & 2 & 1 \\ 2 & 3 & 2 & 0 & 2 \\ 1 & 2 & 1 & 2 & 0 \end{pmatrix}$$

the corresponding distance matrix N is symmetric with elements (a, b, c, d, e):

$$N = \begin{pmatrix} 0 & & & & \\ 81 & 0 & & & \\ 100 & 90 & 0 & & \\ 83 & 78 & 81 & 0 & \\ 90 & 83 & 88 & 84 & 0 \end{pmatrix}$$

The clustering process continues in a bottom-up fashion. It proceeds in a series of successive fusions of the n objects into clusters, to reduce the size of the distance matrix. The grouped objects are those with the smallest distance (therefore, those whose strength of coupling is higher, because they have the largest number of common data bindings). From the initial matrix N the objects b, d are the ones to be grouped. Subsequently, they are fused with the nearest groups/objects. The distances between the newly created cluster (b, d) and the objects a, c, e are calculated as follows:

$$N_{(b,d),a} = \min(N_{b,a}, N_{d,a}) = 81$$

$$N_{(b,d),c} = \min(N_{b,c}, N_{d,c}) = 81$$

$$N_{(b,d),e} = \min(N_{b,e}, N_{d,e}) = 83$$

The group (b, d) is considered as a single object whose distance to other objects in the system is defined as the distance to the closest element of the group. After the first iteration the matrix becomes N_1 with elements a, (b, d), c, e:

$$N_1 = \begin{pmatrix} 0 & & & \\ 81 & 0 & & \\ 100 & 81 & 0 & \\ 90 & 83 & 88 & 0 \end{pmatrix}$$

During the second iteration cluster (b, d) is grouped with objects a and c at level 81 forming a new cluster $(a, c, (b, d))$. Note that a and c are drawn into the (b, d) cluster, even though they have no bindings to each other. The distance matrix at the latest iteration becomes N_2 with elements $(a, c, (b, d))$ and e.

$$N_2 = \begin{pmatrix} 0 & \\ 83 & 0 \end{pmatrix}$$

Finally, object e is fused into cluster $(a, c, (b, d))$ at level 83. Graphically, the created clusters are depicted as a dendrogram in Fig. 1.

Mathematically, cluster analysis has some weak points. It is based on limited knowledge of the objects (i.e., some specific object characteristics) and it is not based on very sound probability models. The counterargument to the latter is that if the classification pattern for characteristics of the measured objects is reasonable, the chance that the taxonomy leads to meaningful results with respect to these

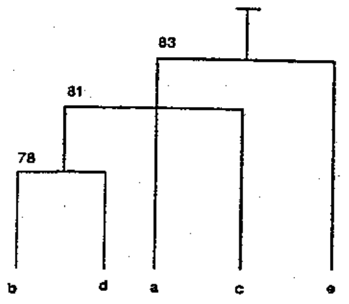

Fig. 1. Sample system dendrogram.

characteristics, is increased [17]. In this study the introduction of Ada Binding Components, along with the definitions of data bindings, offers a clearly defined classification scheme for measurement of data use and visibility. It is also worth mentioning that in our analysis and in agreement with the above presented example we have not used any weighting scheme for the various data bindings.

4.1. *An example*

A small example is shown below. The package EXAMPLE contains 4 subprograms, BAR, COO, FOO and ZOO. ZOO is the entry point for the package.

```
package         EXAMPLE is
                procedure BAR(X : in INTEGER);
                function COO(Y : in INTEGER) return INTEGER;
                procedure FOO(Z : in INTEGER);
                procedure ZOO ;
end EXAMPLE;
package body    EXAMPLE is
                A, B, D     : INTEGER;
                C             : INTEGER :=0;
                procedure BAR(X : in INTEGER) is
                begin
                                C := X + B;
                end BAR;
                function COO(Y : in INTEGER) return INTEGER is
                                TEMP : INTEGER := 0;
                begin
                                TEMP := Y + C;
                                return (TEMP);
                end COO;
                procedure FOO(Z : in INTEGER) is
                begin
                                D := Z;
                                A := COO(D) + C;
                                BAR(A);
                end FOO;
                procedure ZOO is
```

```
                    begin
                                        D := 1;
                                        B := 2;
                                        FOO(D);
                    end ZOO;
begin
null;
end EXAMPLE;
```

According to the definitions, the data bindings found in this piece of code are: (ZOO,D,FOO), (ZOO,B,BAR), (FOO,A,BAR), (BAR,C,COO), (FOO,D,COO), (BAR,C,FOO), (FOO,COO,COO). The last one of the above bindings is due to a returned function value. Figure 2 illustrates the clustering tree of the package ABCs. COO and FOO are the most tightly coupled and are clustered first. BAR is grouped next and finally, ZOO comes in to complete the system.

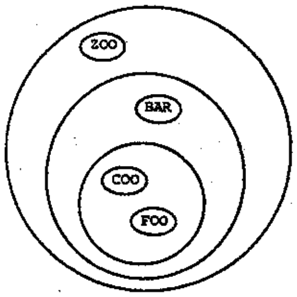

Fig. 2. Clustering structure.

5. Design Outline of the dbt Tool

The basic computations to be performed by the data binding tool are: 1) Identification of Ada data Binding Components (ABCs), 2) Computation of Data Bindings among the various ABCs, and 3) Application of the Mathematical Taxonomy method. The first two functions are performed simultaneously when the source code of an Ada program is parsed. Their output consists of the data binding matrix M whose size is equal to the number of identified ABCs.

The tool has been developed using an Ada grammar adapted for Lex [26] and Yacc [22] specifications. The generated compiler was used to parse the input programs. The challenge was that implementing source code metrics using Yacc specifications alone is not sufficient for complex structured languages such as Ada. Intermediate language representations are necessary for an efficient measurement process. The definitions of bindings and the handling of procedure and function calls require information about the use of globals and the association of program entities to correspondent ABCs. Therefore, a pertinent symbol table for the parsed program needs to be created. The main idea behind the design of the tools is that the Ada source program be transformed into an "intermediate" representation, which is comprised of an interconnected set of tables, called *frames* or *cells*. Each lexical scope maintains such a frame. Every subprogram, nested subprogram and package has its own frame in the structure of this intermediate representation. Frames are connected according to their scope position in the program. The resulting structure is memory resident. The format of a frame is as follows:

```
struct abc_frame          {
        char              name[MAX_LEN];
        char              *type;
        char              *returns;
        int               _num_id;
        tbl_of_parms      _in;
        tbl_of_parms      _out;
        tbl_of_parms      _in_out;
        tbl_of_vars       loc_vars;
        tbl_of_vars       exp_vars;
        type_desc_struct  loc_types;
        type_desc_struct  exp_types;
        struct abc_frame  *subord;
        struct abc_frame  *super;
        struct abc_frame  *next;
        struct abc_frame  *prev;
        struct abc_frame  *ren_frame;
        spec_purp_tbl     special;
        };
```

Each frame maintains information about the ABC it describes. Among others, it includes the following: the name of the elaborated ABC, the type of the frame (i.e., subprogram, package, gen_package, function, etc.), a unique numeric frame identifier _num_id which is used by the searching routines of the tools. Navigation pointers that assist in traversing the structure are provided such as *subord, *superior, *next, etc. For example, *subord is a pointer providing access to ABCs nested in the current component. Every frame maintains a list of the formal parameters (wherever applicable) along with their types (_in,_out,_inout). Information about local types is kept in a list (pointed to by loc_types). Types visible from the outside of the frame environment are pointed to by exp_types. The handling of local (loc_vars) and exported (exp_vars) variables is done similarly. Variables are depicted by an object name and their type descriptor. For example, the intermediate representation for the package described below is shown in Fig. 3:

```
package COMPLEX_NUMBER is
        type COMPLEX is private;
        function "+"(X,Y:COMPLEX) return COMPLEX;
        function "-"(X,Y:COMPLEX) return COMPLEX;
        function "*"(X,Y:COMPLEX) return COMPLEX;
        function "/"(X,Y:COMPLEX) return COMPLEX;
        private type COMPLEX is
```

```
                record
                REAL_PART, IMAG_PART: REAL;
                end record;
end COMPLEX_NUMBER;

package body COMPLEX_NUMBER is
        function "+"(X,Y:COMPLEX) return COMPLEX is
        begin
        return(X.REAL_PART+Y.REAL_PART,X.IMAG_PART+Y.IMAG_PART);
        end;
....
end COMPLEX_NUMBER;
```

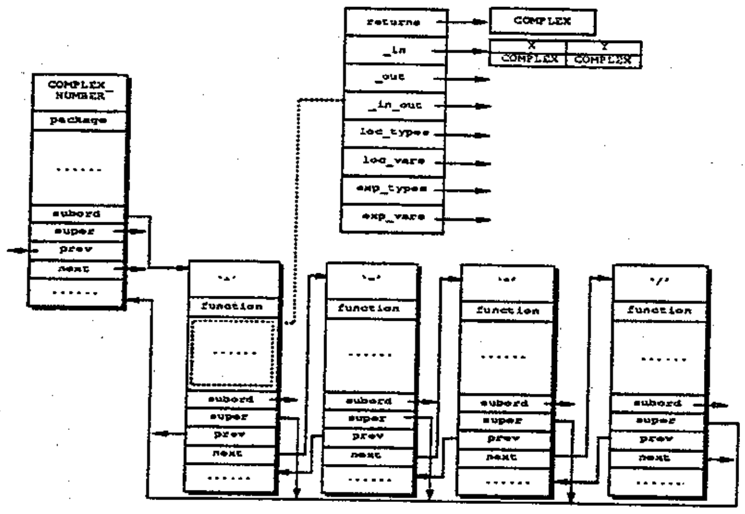

Fig. 3. Intermediate representation of an Ada package.

A reasonably sized Ada program consists of a number of packages and subprograms compiled in a predefined order to produce object code. So far, only the description of simple ABCs has been discussed. The "partial" representations of packages and subprograms need to be connected in a way that establishes visibility via `WITH` and `USE` clauses. A top level structure that provides the capability to join the structure of the relevant compilation units is proposed at this point, termed the *web*. For example, consider a system consisting of three library units (two packages and one procedure) set up as follows:

```
package A is ....
              ...
end A;

with A ; use A;
package B is ....
```

```
    ...
end B;

with B ;
subprogram main is ...
...
b_proc();
...
end main;
```

Figure 4 shows the top level structure and how visibility is maintained. This web permits the identification of ABCs coming from different compilation units. For instance, procedure b_proc() invoked in main establishes bindings between these two involved ABCs. In order to compute the number of data bindings, the frame of b_proc() needs to be found. Package B is searched first as the most likely unit to contain b_proc(), because main is WITHed to B. If not found there, the next WITHed package is examined. Since package B is also WITHed to A, package A is the next place to look for b_proc(), etc. The tool contains the appropriate search routines to navigate through the structure and identify the correct frame within a hierarchy of representation frames.

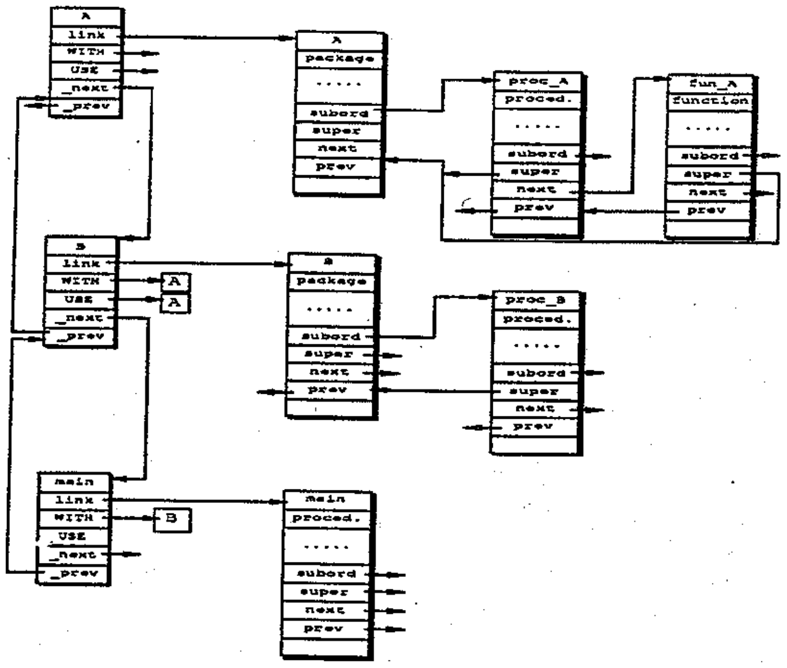

Fig. 4. Top level data structure (*Web*).

In general, the frame of an invoked ABC is required to be found before the counting of data bindings commences. For every formal parameter of type *IN* or *OUT* one binding is counted. Two bindings are counted for every formal parameter of type *IN_OUT*. The former parameters provide a unidirectional way of information flow; the latter gives a bidirectional information flow. While parsing the body of an ABC, a record of what is being either referenced or assigned is maintained. The goal of this record is to assist in the computation of bindings especially when global variables are involved. At the end of every library unit parsing, data bindings due to globals may be determined. Unresolved references and assignments are stored to be resolved at the end of the program parsing when information about all units is available.

The complex structure of Ada causes several problems in this phase of the tool design. Some of the more important ones are: separate compilation, renaming, generics and overloading.

The tool accepts as input all compilation units described by the name of the files they reside in. Files are opened and closed in the order given in the command line. It is also assumed that files are given in the correct compilation order. The problem that still remains to be solved is that of subunits. A rather simple approach to overcome it would be to pre–process and expand the source code of the program with the subunit bodies. Another approach, lazy subunit elaboration, would be to postpone the processing of subunits until their files are encountered. Whenever an ABC declaration is parsed the tool sets up its corresponding frame. The information given in the `separate` clause along with the name of the ABC (both found in the stub file) assist in tracking the ABC's frame in the program representation structure. While the body is being elaborated, bindings due to ABC calls can be easily derived. On the other hand, bindings due to globals require complicated processing since lists of assigned and referenced objects need to be kept even after the end of parsing of library units. The first solution (of code expansion) was adopted as more natural and easier to implement compared to the lazy subunit elaboration.

Renaming can be applied to variables, exceptions, subprograms, task entries and packages. Renaming of variables can be accommodated by having pointers to the structure of the renamed object. Renaming of subprograms and task entries as well as packages can be handled in a similar way. For a renamed ABC, a new frame is set up in the system representation of type "renamed" and the field `ren_frame` points to the renamed ABC (otherwise it is `NULL`). The cell is created within the scope where the renaming was encountered.

Generics are kept in a separate structure, where explicit references about the imported types and subprograms are maintained. Since bodies of generics are elaborated before instantiation, the tool keeps a record of which imported subprograms are invoked in the bodies of the generic. The field `special` of the template keeps this information. Imported ABC frames (i.e., imported functions and subprograms) are designed to be at the same lexical level as that of the generic unit in the structure. Figure 5 shows how frames are set up during the elaboration of the generic

FORMATTER package. The outline of the package is:

```
generic
       type WORD is private;
       type LINE is private;
       with function VALUE_OF(THE_WORD : in WORD) return STRING;
       with function IMAGE_OF(THE_LINE : in LINE) return STRING;
package FORMATTER is
       procedure SET_UP;
       procedure APPEND ( THE_WORD : in WORD;
                          THE_LINE : in LINE );
       procedure CRE_DOC;
end FORMATTER;

package body FORMATTER is ...
       ...
end FORMATTER;
```

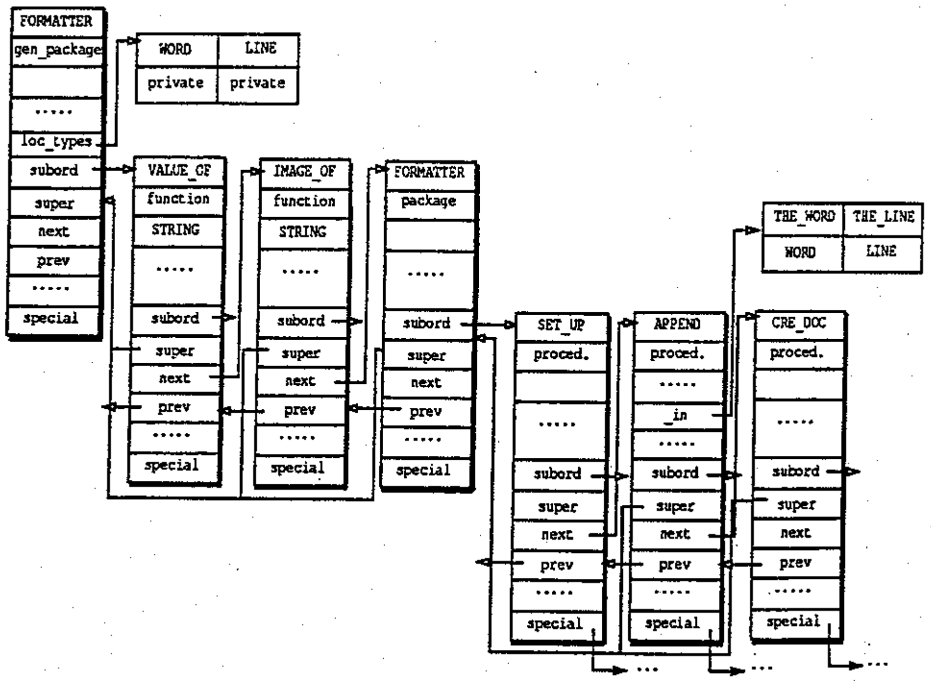

Fig. 5. Frame description of a generic package.

The instantiation of a generic unit is basically a copy of the frame constructed in the generic form having imported subprograms and types changed accordingly. Upon a generic instantiation, bindings among the instantiated ABCs and the imported ABCs can be easily computed. Bindings occurring among the instantiated ABCs and the rest of the system's ABCs are calculated whenever needed thereafter.

Thus, the instantiation of the FORMATTER package

```
package INST_FORMATTER is new FORMATTER
        ( WORD => WORD_STR, LINE => LINE_STR,
         VALUE_OF => A.VALUE_OF, IMAGE_OF => A.IMAGE_OF);
```

creates a copy of the structure starting from the FORMATTER frame (prev is set equal to NULL). This substructure has all the types changed to WORD_STR and LINE_STR. Since imported subprograms are visible at this point, data bindings can be collected right away. This is possible because the information for subprogram invocations is kept in the structures described by the special fields.

Overloading is perhaps the most challenging problem to be dealt with in the design of an Ada based tool. Type information collected throughout the parsing phase and stored in the *intermediate representation* of an Ada program is used to disambiguate the invocation of overloaded ABCs. A variation of the algorithm proposed in [12] is used to achieve resolution given that the proposed representation resembles a *decorated* tree. The current version of the tool does take some effort to disambiguate overloaded ABCs, but is rather limited. However, this does not affect the validity of our experiments since the examined systems did use very limited overloading.

6. Use of the dbt Tool

In this section, we first review a number of issues on reusability and discuss the examined design methodologies. We set goals to be reached through the use of the tool and present the conclusions of our experiments with a set of Ada systems.

6.1. *Issues in reusability and examined design methods*

It has long been recognized that reusable components assist in the construction of quality software and in the avoidance of the late delivery problem [35]. Reusability may occur as soon as a computing environment is well understood by its system designers and/or programmers. This maturity comes with time and with better understanding of the way people conduct their system development efforts. In this paper, we are concerned with the reusability of products and in particular with that of the source code. The fundamental question is what can be reused from an existing software system and how easy it is.

Modular designs have been proposed [11] as a flexible means to promote reusability. Booch [8] claims that greater gains are obtained from the reuse of archived designs and subsystems than from individual components or objects. This does not occur unless there is some specialization in the problem domain. For example, consider a software factory: although there might be some diversity in the type of the developed software, the domain of the applications is not expected to change often and drastically. In this case, certain software component categorizations can result into systems whose many functionalities are reused continually and in a rather controlled way [31]. The degree of deviation of an undertaken project from the major domain of development in such an organization dictates the granularity of reused objects.

A software component/subsystem cannot migrate to another environment without its operational context. The operational context is defined by the data structures that are either accessed or assigned by the component in question and the *token data structures* that are being passed from one component to the other in the form of parameters. A major problem with this type of software construction is that a change in the data structures may send a ripple effect throughout the number of elements to be reused. Object–Oriented-like systems similar to those developed in Ada partially prevent this problem. Although a set of data structures along with its manipulating routines usually form a good reuse object, understanding its interaction with other components is equally important to its extraction.

A reuse process cycle consists of the following activities: 1) *Search*, which tries to find the candidates for reuse, 2) *Identification*, which deals with naming, external references, isolation of the involved data structures and the operational space, and 3) *Qualification* which performs careful semantic analysis such as finding the support packages, functionality, test cases, etc. This study basically provides a novel way to assist during the first step of this process model and partially during the second using the combination of data binding metric and cluster analysis. Data bindings provide a way to quantify the interaction of components with each other either through a direct (token passing) or an indirect fashion (through common data structures).

Having even a small size system (less than 5K lines of code), it is rather tedious to browse through and try to find what can be reused and in what way. The output of the data binding tool–as it is shown later–offers two opportunities. First, it gives an abstract, yet accurate glance of the potentially reusable objects and second, it partially reveals the scope of the operational context of an object. With a number of experiments performed on Ada systems, a possible classification of the produced dendrograms can be derived. The correlation of this classification scheme to both *Search* and *Identification* phases could serve as an extremely useful tool for the isolation of the code to be reused. Dendrograms alone cannot help during the qualification phase though. A metric oriented approach for this phase is presented in [10]. However, it is the general belief that the qualification phase requires a lot of human intervention [25, 3]. The measurement based technique given in [10] could be improved considerably if the *basic reusability attributes* were enhanced with more diverse metrics [14].

Design involves a substantial effort to reduce complexity and facilitates the resolution of implementation problems in a system under development. The design process involves the transformation of requirements into a model of the software system upon which the implementation may be based. The design model may be analyzed for qualities such as functionality, performance, reliability and maintainability to provide an indication of problem areas. Based on these analyses, corrective action can be taken at an early stage. Several design methodologies have been proposed over the years. Most of these techniques can be classified as either top–down decomposition (TDD) or bottom–up composition (BUC).

The essential theme in top–down decomposition methods is that systems are decomposed into a set of cooperating parts, with each part at the same level of abstraction. The design at each level hides the details of the design at lower levels, since only the data and control flow across the components of that level are depicted. If needed, this decomposition process is applied as many times as is appropriate. The details of the design of low-level components are thus postponed until the last stages of design. Issues like implementation feasibility and ability to manage the components decide the level of nesting. Functional decomposition is an example of a top–down, hierarchical method. Here, the designer separates the system into its top level functions. Each of these functions is similarly decomposed until the appropriate level of decomposition has been reached. Breaking up the system according to functions is only one method for decomposing a system. The Jackson System Development technique recommends a top–down decomposition of the system according to data structure [11], and the Yourdon–Constantine Structured Design recommends a decomposition according to data flow [38]. The common idea in these methods is that the refinement is done in a top–down manner, which focuses attention on designing solutions to the whole problem before concentrating on designing solutions to the sub–problems.

On the other hand, bottom–up design techniques embrace the idea that the designer's primary concern is the development of the elementary system's units. The designer determines the most critical units based on experience, intuition or a simple analysis. These parts are the focus of the process, and the remainder of the design is tailored to accommodate the design of these critical parts [34]. Hence, certain virtual machines and abstractions can be constructed. A rapid prototyping method often utilizes BUC, to allow for the early development of the critical parts, and for an early assessment of the design feasibility. These techniques are generally not recommended alone, but if they are to be used, they should only be used to investigate the critical parts, followed by a top–down approach for the final design [34]. BUC is appropriate where the elementary units are of primary concern, such as in the development of sets of utility packages. There seems to be a relationship between BUC and reuse–oriented design, in that the focus is on the elementary units, and how these units can be integrated to generate the desired system.

Another technique that has received a lot of attention recently is the paradigm of Object–Oriented Design (OOD). The main idea is that instead of trying to partition the system according to architectural, functional or informational boundaries, the system is structured around objects. Each system module stands for either an object or a group of objects along with their operations in a problem subspace. Objects are extracted from a model of the real world problem that needs to be automated. The application of Object–Oriented design is accompanied by the use of abstract data types. OOD has been influenced by the techniques used in Object–Oriented programming. Broadly speaking, an OOD system could be categorized as such if the following attributes are present [24]: (1) objects, (2) object classes, (3) inheritance of properties of one class to its subordinates, (4) dynamic binding, or the association of the correct code to be executed at run time, and (5) polymorphism, or the ability of a reference to be bound to more than one class object instance over time.

We need to recognize that Ada is limited in terms of its support of these attributes. More specifically, objects and classes are directly supported by the language constructs and its design rationale. Objects are the run–time elements that depict the real–world entities. Classes are sets of possible objects. It can be advocated that generics offer support for classes. The rest of the properties are not directly or implicitly supported by the Ada rationale. Ada is predominantly an "encapsulation language", but as was pointed out, it maintains a certain number of characteristics that enable pragmatic Object–Oriented design [8]. In most cases, a set of fundamental system objects that model a problem are identified. Subsequently, their operations, interfaces, interactions and structures are designed. In contrast with the previous two approaches, this methodology tends to be more "localized" as far as changes are concerned, since extensive use of abstract data types and state machines is advocated. Constructs such as the Ada package make this type of design feasible, as localized data structures, invisible types and their implementations are supplied. Generics are another means for facilitating OOD, by furnishing parameterized object templates.

Many design techniques borrow concepts from several of the above classes (i.e., TDD, BUC, OOD). These concepts and the way they are combined distinguish the variants. For instance, incorporation of a reuse oriented process into an object–oriented design paradigm will also affect the resulting process model, as the designer will be concerned with both the objects that are modeled and the objects that are stored within the repository. Therefore, when assessing design, one has to consider how the techniques are combined.

6.2. *Source reusability and design assessment goals*

Several guidelines have been proposed on how to reuse software product components [10, 35] and how to assess design in a practical manner [21]. The Goal/Question/Metric paradigm (*G/Q/M*), as discussed in [2], provides such a framework for evaluation of products and processes viewed from different perspectives. The main idea is that in order to evaluate factors (goals) we need to refine them into a set of questions. Those questions deal with concepts and ideas on how to achieve these goals. In the next step down, the framework's questions are further decomposed to metrics. Metrics are designed around measurable entities which affect either the reuse attributes or the design process and they may present the means for the automated measurement. Results of metrics can be examined to answer the questions asked and thus, to validate the quality factors at the first level. The validation of the *G/Q/M* decision tree is carried out in a bottom–up way.

The goal to reuse software objects could be analyzed to a number of questions that likely qualify some of the existing ones as candidates [13]. Some of them could be stated as follows:

- What is the distribution of the several ABCs of a library unit throughout the system dendrogram?
- Which library units are involved in the formulation of the several dendrogram clusters and what is the reason for that?
- Where do ABCs which interact with their outer environment appear on the dendrogram?
- Does a particular dendrogram have a specific cluster formulation that allows reuse?
- Does the majority of clusters contain ABCs from all the library units and if so what are the implications for the code to be identified and reused?
- If the user is familiar with the code and there are some discrepancies between what the dendrogram presents and what is expected, what are the possible (if any) explanations for that?

The distribution of the ABCs of a library unit into several clusters is used as a way to recognize their operational context. For instance, if a cluster contains elements from three library units then the operational context of each of these units can be found very easily by looking up the way these units are WITHed to each other. The indirect interaction of different ABCs through library units (with visible entities to all interested parties) is also an integral part of the dendrogram layout. Isolating independent subtrees in a system dendrogram is of great importance to reuse. That would facilitate the exploration for stand–alone source code in an existing system. Another question to be answered would be if different dendrogram formulations give varying degrees of reusability. By default, some parts of software systems are written in such a specialized and rigid manner that is not possible to be re–engineered. Would it be possible to easily isolate such (sub)systems within a dendrogram? Finally, people who maintain some knowledge about the system would be extremely interested to see a similar picture of the system on the dendrogram. If this is not the case, they should try to explain this irregular behavior and its implications on the system's reusability.

Design assessment is used to obtain an indication of the product quality, allowing for corrective action to be taken if necessary. Goals of a quality modular design include inter–module independence and intra–module integrity [29]. A well rated design should present a balance of these two major quality factors. Questions that can be asked for the above mentioned quality factors include the following:

- What is the relative coupling among ABCs?
- How cohesive are the involved ABCs?

- Are there components with excessive external control flows?
- Are the interfaces defined at the appropriate level of abstraction?

Coupling measures the interaction between different software or design components. By minimizing the interaction among different entities, a greater degree of independence can be achieved. This independence helps to achieve the minimization of the effect of a change. We identify two types of coupling, data coupling and control coupling, dealing with inter-module data flows and control flows, respectively. More specifically, data coupling appears whenever there is an interaction of two or more components via a data element, while control coupling (unlike the definition in [29]) can be found if there is transfer of control between two elements. **dbt** has the capability to collect all data and some of the control couplings, thus, providing a view of the system focused on its coupling.

Cohesion measures the relative strength of a component, or how well related the internal objects are to one another. Using the **dbt**, we can obtain measures of two types of cohesion, data cohesion and control cohesion. Data cohesion is defined as the degree to which the internal objects utilize the same data elements (similar to the communication cohesion defined by Myers), while control cohesion measures the relative proximity of the internal objects invocation times (much like Myers' temporal cohesion). Control cohesion can be found in modules that execute predominantly in their locales. The call tree tool (**ctt**) reported in [14] detects higher-level ABCs (package level and above) with control coherence by identifying the location of the ABCs' subcomponents in a call tree. If they usually appear together in the tree, one can conclude that the ABC exhibits control cohesion. Examination of the system dendrogram can similarly provide indications of the degree of data cohesion in a higher-level ABC. If the components of a particular ABC are clustered together at an early stage, we may conclude that the ABC in question is data cohesive. If, however, these components are grouped in separate clusters then the ABC does not exhibit data cohesion.

In the design of the components, it is desired that they have a clear, focused purpose. At the lower, or subprogram level, an estimate of the modules' integrity of purpose can be obtained by examination of its fan-out. Those with excessive fan-out values may have too broad a purpose, and perhaps should be further decomposed. **dbt** identifies ABCs with fan-out exceeding a threshold, facilitating further investigation. Evaluation of interfaces is essential in assessing design by deriving outliers which use a great number of interface items that lack abstraction and need to be redefined.

The **dbt** furnishes the means for the analysis of both data and control coupling. By providing the capability to analyze modules of varying granularity, we can determine the interaction of subprograms within a package and the interaction between packages. Taking this one step higher, by grouping packages and procedures into subsystems, one can analyze interaction within and across subsystems. This is how we see this tool being utilized for design assessment.

6.3. *Test data*

The software used in the analysis was supplied by the RAPID Center Library project and the Software Engineering Group at the University of Maryland. The RAPID Center Library consisted of 13 sets of Ada compilation units making up 15 systems of various sizes. The University of Maryland software contains 3 sets of Ada compilation units making up 4 systems. Their sizes range from a few hundred lines to approximately five thousand lines of source code. Reuse is a primary objective for the RAPID Center. Therefore, all of their systems were designed and implemented for reuse. The three major design techniques outlined in a previous section were utilized for the development of these systems. Programs provided by the Software Engineering Group were developed using predominantly Object-Oriented design.

6.4. *Description of tool output*

In this section we describe the output of the tool. The **dbt** produced dendrogram is a tree structured representation of the clustering of system components (ABCs). A component in the dendrogram has three properties associated with it:

- its level in the tree
- the subtree to which it belongs
- the number associated with the cluster.

The dendrogram produced for one of the analyzed systems named *String_Utilities_Package* is shown in Fig. 6. There are nineteen components, four levels, and six unique cluster numbers.

The numbers represent the passes at which components were clustered. The six unique cluster numbers indicate six iterations in the clustering process. The components with the smallest number were clustered during the first pass of the clustering. The number (divided by 100) associated with a cluster at every pass represents the probability that a data binding chosen from the set of bindings that involve an element of the cluster is not a binding among the components of the cluster. Thus, the components *Leading_Nonblank_Position* and *String_End_Position* represent the most tightly bound components in the system, with a .50 probability that a binding to or from either one is to the other.

The number 0 associated with a cluster means that there are no bindings among the elements of the cluster. Since this number occurs only at the highest level, it also means that there is no actual data binding to any other components in the system. In terms of data coupling, these components are completely independent of the system. Some of them may present potential data bindings [19].

In the next pass of the clustering process, two new clusters were created (at 66). At cluster number 71 one more cluster containing *String_Equalities* and *Change_Character_Case_Lower* was formulated. Until this point 9 of the 19 components have been clustered into 4 first-level independent clusters, as Fig. 6 depicts. At

```
PACKAGE String_Utilities_Package

100
    0
        Fill_String
        Clear_String
        Next_Blank_Position
        Is_Numeric_String
        Is_Alphabetic_String
        Change_String_Case_Upper
        Change_Character_Case_Upper
   75
       66
           Trailing_Nonblank_Position
           String_Length
           Is_Empty_String
      50
          Leading_Nonblank_Position
          String_End_Position

  80
      Change_String_Case_Lower
      71
          String_Equalities
          Change_Character_Case_Lower
      75
          Substitute_Substring
          Substitute_Character
          66
            Substring_Position
            Character_Position
```

Fig. 6. Example of dbt output.

75, two already existing clusters were bound together and another group containing *Substitute_Substring* and *Substitute_Character* was created and bound with an existing group with cluster number 66 (note that components carrying out similar computations or performing operations on the same data are grouped in the same cluster). This creates two second-level clusters containing 11 components.

The numbers in a particular pass of the clustering process are defined in a different context than the numbers in the prior pass (because of the transformation made), thus, numbers do not have a consistent meaning across passes. However, there is a partial ordering based upon bindings, defined among the clusters at the same level. For instance, *Leading_Non_Blank_Position* and *String_End_Position* are more tightly bound than *Trailing_Nonblank_Position*, *String_End_Position*, and *Is_Empty_String*.

The subtree to which a component belongs defines its clustering subgroup. For example, the *Substring_Position* and *Character_Position* cluster is grouped with the components *Substitute_Substring* and *Substitute_Character*, creating a dendrogram subtree (low part of Fig. 6). Finally, all the clusters coalesce into one clustering group, representing the entire system.

6.5. *Interpretation of dendrograms*

Dendrograms are used to get an assessment of the strength and the coupling of the various clusters in the system. Dendrograms can be examined along the following guidelines:

- For any level n, each of the clusters at level $n-1$ forms a subsystem usable for building n and higher-level functions. If the binding within a cluster is strong, (i.e., the cluster number is small relative to the cluster number of the level n) then this cluster exhibits close coupling among its components. This cluster would also be evaluated as having loose coupling to the other clusters at that level.
- If, however, the binding within a cluster is weak, (i.e., the cluster number is close to the cluster number at level n) then the top-level components of this cluster do not exhibit tight coupling, which means that the cluster does not have high strength. Also, its coupling to any other component at that level is weak. This would imply that even minor changes to the system might change the dendrogram structure with regard to the components in this cluster.
- In general, it could be said that for any level n of the dendrogram, the individual components at level n are auxiliary. They use clusters at $n-1$ as "core" components to build on. They add functionality to the system at level n and may themselves be used as building blocks for level $n+1$. If the number of siblings at every level of the dendrogram is small, e.g., at most 2, independent of the depth or cluster numbers, then a highly nested structure has been encountered.

6.6. *Analysis of reusability results*

Examining each of the Ada systems from the reusability point of view, we analyzed the types of extensions as well as the potentially reusable subsystems that are available for future system development. Consider the dendrogram of Fig. 6. At the top level, each one of the clusters is independent. Any of the 0, level components may be deleted or new ones may be added without changing any of the existing clusters (unless some interaction with the ABCs of these clusters is introduced). Under the assumption of no control coupling[a], 0–level ABCs are there only to support potential user services and, from the point of view of the existing system, are not needed. This provides a simple but still fundamental basis for enhancing or reducing the functionality of a system. For example, the function *Is_Numeric_String*

[a]Control couplings can be easily found using the ctt tool [14].

can be deleted without any effect on other components. Therefore, package size can shrink if that function is not required. Also, new components that use any component from the clusters at level 75 and 80 may be exclusively added to the system to enhance its functionality without worrying about the effects on the other clusters. These components define a class of components that can be easily added into a system.

Since each of the clusters at level 75 and 80 are also independent of each other and do not use any of the ABCs from the 0 level, they themselves form reusable subsystems. Fig. 7 provides a cluster view of the system of Fig. 6. The notation offered by this view can be explained as follows: the group depicted by 71 contains two ABCs namely, *String_Equalities* and *Change_Character_Case_Lower*. A design that offers a large number of clusters at the top level provides a large set of independent ABCs that can be used as the basis for developing other systems. In total, RAPID offers fifteen reusable systems with a total of 31 reusable subsystems while 4 UM systems offer 7 reusable subsystems.

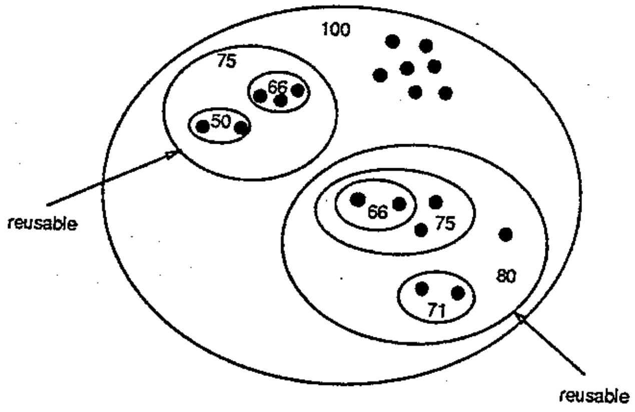

Fig. 7. A view of the clustered system.

Although we noticed that ABCs from relevant library units were grouped in the same or related clusters, we observed some discrepancies in the distribution of some ABCs in the system dendrogram. For example, one ABC from a library unit would only cluster with another set of Ada data Binding Components, while the rest would cluster somewhere else. We found two explanations for that: either there was heavy implicit interaction through a third library unit, or the number of the ABC parameters (*token data structures*) was extremely large. The latter affected the outcome of the clustering process and produced the above unexpected result. No abstract interfaces are responsible for such unusual distributions of ABCs.

Our attention turns now to the classification of dendrograms and their characteristics in the context of reusability. Groups of ABCs — as mentioned before —

are identified by rating the three characteristics mentioned above: depth, cluster number, and number of sibling subtrees. *Plateaus* represent groups of ABCs which constitute either sets of stand–alone components or groups where it is extremely difficult to isolate individual reusable candidates. Low and high types of plateaus were identified throughout the experiments.

Low plateaus demonstrate small cluster numbers (less than 20). Their depth is shallow. Low plateaus suggest components that perform computations with intense interaction with the rest of the ABCs. They correspond to small–sized computational tasks which are well understood and are called a lot of times from the upper–level ABCs. Therefore, they can be reused without major rework. A set of zero cluster number plateaus in a system made of reusable code, reveals ABCs of negligent role in the viability of the new system as a whole. Thus, they could be eliminated. In certain situations, it was noticed that such plateaus constitute half the code of utilized packages. Some of the functions of the system depicted in Fig. 6 belong to this category. Components such as *Fill_String*, *Next_Blank_Position*, *Is_Numeric_String*, etc., appear in the system listing but are never called and so they never establish bindings with the non–zero cluster numbered ABCs.

In contrast, high plateaus are characterized by high cluster numbers (in the range of 85–100) and a large number of sibling subtrees. Their depth is usually shallow. High plateaus are formulated when low-level logical components are heavily used in a system. Lower-level ABCs of such a plateau are essential for the computations performed by higher-leveled components. Reusability of a small depth ABC needs to be accompanied by those heavily used low-level components. High plateaus represent systems that do not prompt "clear" groups of ABCs. Poor module design is probably to be blamed for it. Three systems were found to present plateau properties from the examined software.

Vertically revolving clusters describe systems structured in a very nested fashion. Their depth is deep. Cluster numbers fall almost always in the whole range (20–100) and the number of sibling subtrees is definitely small. Consequently, isolating reusable subsystems is impossible and reuse is limited to the system as a whole. This is due to the fact that there is certain difficulty in isolating portions of the code from its environment. Two systems presented clearly nested structure. Such systems are used "as provided" and need major rework in the case of intended partial reuse.

Regularly revolving systems encompass most of the examined software. Their structure is around a major cluster that undertakes the role of the *main* component in the system. They are organized in a multiple (but not extremely deep) level fashion. Typically, they maintain a large number of sibling subtrees and their cluster numbers fall in the range of 40–100. When reusing a cluster from such systems, it is absolutely essential that all the contained cluster components migrate to the new environment. Besides, it was noticed that certain pieces of sibling clusters (due to the partial ordering mentioned earlier) may be required.

6.7. *Design assessment results*

Designing with one of the top–down methods produces systems with dendrograms resembling the functional decomposition that was followed. The components of a module predominately exhibit a high degree of data and control flow amongst themselves, implying that they are data and control cohesive. At a higher level, coupling is minimized, and is used to present the means for interconnection among groups of ABCs, providing a clear separation of component functionality. Although an exact match between the trees and the design decomposition was not found, it was possible to identify the general trend followed by the decomposition process. To summarize, for the TDD systems, if there is close resemblance between the decomposition and the tree, then the reviewer may have greater confidence in the quality of the design.

Systems created with the bottom–up techniques tend to have fewer levels of depth and therefore exhibit a flatter structure. In most cases, BUC systems start either with partial reuse of already existing systems or with exploratory development of low-level functionalities. The remaining parts of the systems are built around these pieces. From the analyzed software, we have seen that such systems are in all examined cases of utilitarian purpose (utility subprograms). Examination of dendrograms showed that cohesive objects were supplied at the bottom-level furnishing groups of these utility ABCs. Increased coupling was found among the higher-level ABCs. One of the difficulties in designing a system bottom–up is the excessive growth due to unused code. As mentioned before, **dbt** clusters, at zero level, all ABCs not involved in data flow, and enumerates all actually called components. These two results determine unused subprograms within a system. Given that typically the size of an Ada system is rather large, eliminating such code would significantly decrease compilation times.

Object–Oriented systems tend to be built as sets of packages located in layers. It is also expected that these layers are relatively flat, in the sense that they do not contain a great deal of language construct nesting (i.e., packages within other packages, etc.). The findings for systems designed with this method are much different from those described above. Examination of the trees for such systems reveals low data and control cohesion (of package level ABCs). However, inspection of the packages in question showed that they are "conceptually cohesive", meaning that they consist of logically related entities.

Dendrograms of such systems group these sequences of procedure calls that carry out semantically related computations through the levels of abstraction. Figure 8 shows such a system with three layers and four objects. Each of them consists of a number of ABCs. The encapsulating bubbles indicate the clustered ABCs as they were generated by the **dbt**. These clusters go across the borders of the object in the system hierarchy. This indicates that the system was designed around conceptually coherent objects rather than minimization of coupling.

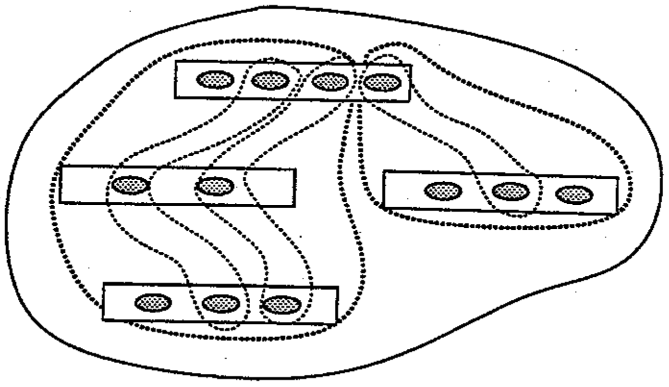

Fig. 8. Clustering across object boundaries.

Unlike TDD and BUC systems, where there was a clear tendency by **dbt** to cluster low-level call tree components before high-level ABCs, OOD systems show a greater mix in their groups. Both high- and low-level components were clustered early by **dbt**, which implies a more even distribution of the computational effort across the system.

On the other hand, a small number of systems produced trees that did not comply with the followed hierarchical methods (TDD and BUC). Possible explanations for these deviations include:

- Many parameterless calls: To utilize such techniques for carrying out a design may be valid but the **dbt** is unable to establish bindings (due to nonexistent data interfaces). Therefore, the clustering gives an altered picture of the system. The call tree tool **ctt** [14] could give a precise picture of such a system since its very role is to identify sequences of subprogram calls. In such a system, use of bindings does not reveal the accurate decomposition and there is certain discrepancy between call trees and dendrograms.
- Use of large global data structures: This is a direct result of the large number of implicit data bindings derived from the interaction of the system ABCs with a particular data area. Figure 9(a) illustrates such a system. Solid lines indicate control flows among ABCs and dashed lines interaction with a global data store. The conformance between the dendrogram and the decomposition of the system was "altered" due to the large number of bindings established through the common data area.
- Non–abstract interfaces: Their side effect is similar to that of the global data structures. An excessive number of interface items results in many bindings which alter the picture of the system under examination. Many simple bindings may force ABCs to cluster much earlier than would be expected. A possible means of alleviating this problem is to adapt to a weighting scheme for the differently-

typed interface parameters. For the time being, **dbt** has been modified so that it reports components that exceed a threshold number of interface items, but no weighting has been used.

- Poor design performed: As Fig. 9(b) depicts, there exist calls of components from deeper ABCs to shallower ones providing a mix in the procedure/function call ordering. The usefulness of the trees in the understanding of systems–apart from the fact of verifying the poor design–is very limited.

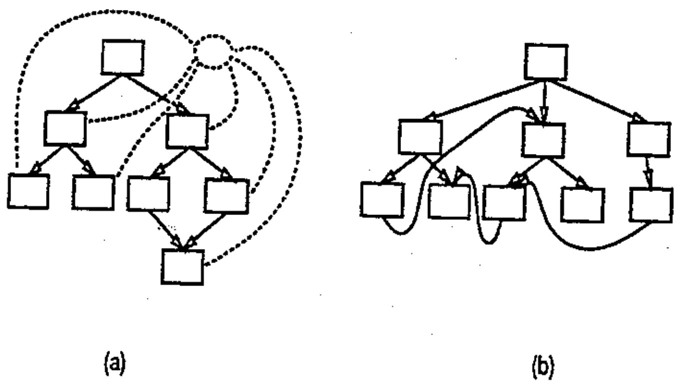

Fig. 9. Irregular system structures.

7. Conclusions

In this paper, we have presented the data binding tool (**dbt**) and demonstrated its use in Ada source code reusability and system design assessment. The tool was built around the metric of data bindings. Data bindings are utilized for measuring the inter–component interactions. The classic definition of the data binding metrics along with the modified one for an Ada environment were presented. We have proposed a flexible scheme for the characterization of Ada components — ABC model. The model allows the user to define the granularity of the examined components at their own will. Elementary components of the model are subprograms, function and task entries. Mathematical taxonomy is used to cluster related ABCs to groups of components, by providing the system dendrogram. The design of the tool was discussed briefly and the major problems encountered were described. The main challenge for the design was to isolate as much information as needed from the source code of an Ada system. The issue of intermediate program representation, particularly that oriented towards measurement purposes, requires further investigation.

The dendrograms were used as the vehicle to reason about reusability, design assessment and in general to understand the way participating ABCs interact. The goals for Ada source code reusability and design assessment were decomposed to quantifiable questions which the output of the tool helped to answer. Several classes of systems dendrograms were identified and types of extensions as well as reusable (sub)systems were accounted for. The implications of different design methodologies used to developed the test software were discussed and explanations for the several types of dendrogram formulations were given. Another important benefit of the tool is that it may be used throughout the design process, so that design can be assessed at an early stage as well as upon completion.

As future work, we plan to examine the issue of a more efficient intermediate representation for Ada systems geared towards measurement, to analyze more complex systems and systems designed with hybrid methodologies, and to adapt a weighting data binding scheme that represents in a more precise manner the established bindings. A prototype of the tool was developed as part of the *TAME* [2] and the *CARE* [10] projects at the University of Maryland.

Acknowledgements

The authors are grateful to the anonymous referees for their valuable comments and suggestions as well as, William Thomas, John Bailey and H. Dieter Rombach for commenting on earlier versions of this paper.

References

1. J. Barth, "A practical interprocedural data flow analysis algorithm," *Commun. ACM*, **21** 9 (September 1978) 724–736.
2. V. Basili and D. Rombach, "THE TAME PROJECT: Towards improvement–oriented software environments," *IEEE Trans. Softw. Eng.*, **14** 6 (June 1988) 759–773.
3. V. Basili, D. Rombach, J. Bailey and A. Delis, "Ada reusability analysis and measurement," in *Proc. 6th Symp. on Empirical Foundations of Information and Software Sciences*, Atlanta, Georgia, (October 1988) Georgia Institute of Technology.
4. V. Basili and A. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Softw. Eng.*, **1** 1 (December 1975) 390–196.
5. L. Belady and C. Evangelisti, "System partition and its measure," *The Journal of Systems and Software*, **2** 1 (February 1981) 23–29.
6. T. Biggerstaff and A. Perlis, Ed., *Software Reusability–Volume I: Concepts and Models*, Vol. 1 of *Frontier Series* (ACM–Press, 1991).
7. T. Biggerstaff and A. Perlis, Ed., *Software Reusability–Volume II: Applications and Experience*, vol. 2 of *Frontier Series* (ACM–Press, 1991).
8. G. Booch, "Object–oriented development," *IEEE Trans. Softw. Eng.*, **12** 2 (February 1986) 211–221.
9. G. Booch, *Software Engineering Using Ada*, Benjamin–Cummings, 2nd ed. (1987).
10. G. Caldiera and V. Basili, "Reusing existing software," *IEEE–Computer*, **24** 2 (February 1991).
11. J. Cameron, *JSP & JSD: The Jackson Approach to Software Development*, IEEE Computer Society Press, (1989).
12. G. Cormack, "An algorithm for the selection of overloaded functions in Ada," *SIGPLAN Notices*, **16** 2 (1981) 48–52.
13. A. Delis and V. R. Basili, "Ada reusability with the data binding tool," in *Proc. 5th Australian Software Engineering Conference*, Sydney, Australia (May 1990) IREE.

14. A. Delis and W. Thomas, "Design assessment of Ada systems using static analysis," in *Proc. Ada–Europe International Conference*, Athens, Greece (May 1991) Springer-Verlag.
15. B. Everitt, *Cluster Analysis*, Heinemann, (1977).
16. C. Hammons and P. Dobbs, "Coupling, cohesion, and package unity in Ada," *ACM Ada Letters*, **IV** 6 (1984) 49–59.
17. J. Hartigan, *Clustering Algorithms* (John Wiley and Sons, 1975).
18. S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, SE–**7** 5 (September 1981).
19. D. Huntchens and V. Basili, "System structure analysis: Clustering with data bindings," *IEEE Trans. Softw. Eng.*, **11** 8 (August 1985) 749–757.
20. N. Jardine and R. Sibson, *Mathematical Taxonomy* (John Wiley & Sons, 1977).
21. R. W. Jensen and C. C. Tonies, *Software Engineering* (Prentice–Hall, 1979).
22. S. Johnson, "Yacc–Yet another compiler compiler," Technical report, AT&T Bell Laboratories (Murray Hill, 1975).
23. L. Kaufman and P. Rousseeuw, *Finding Groups In Data: An Introduction to Cluster Analysis*, Wiley series in probability and mathematical statistics (John Wiley & Sons, 1990).
24. T. Korson and J. McGregor, "Understanding object–oriented: A unifying paradigm," *Commun. ACM*, **33** 9 (September 1990) 40–60.
25. R. G. Lanergan and C. A. Grasso, "Software engineering with reusable designs and code," *IEEE Trans. Softw. Eng.*, SE–**10** 5 (September 1984).
26. M. Lesk, "Lex–a lexical analyzer generator", Technical report, AT&T Bell Laboratories (Murray Hill, 1975).
27. R. C. Linger, H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice* (Addison–Wesley, 1979).
28. B. Meyer, "Reusability: The case for object–oriented design," *IEEE–Software* (March 1987) pp 50–64.
29. G. Myers, *Composite–Structured Design* (Van Nostrand Reinhold, 1978).
30. United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI–MIL–STD–1815A–1983 ed. (February 1983).
31. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE–Software*, **4** 1 (January 1987).
32. W. Royce, "TRW's Ada process model for incremental development of large software systems," in *Proc. 12th International Conference on Software Engineering*, Nice, France (March 1990).
33. R. Selby and V. Basili, "Error localization during software maintenance: Generating hierarchical system descriptions from source code alone, "in *Proc. Conference on Software Maintenance*, Phoenix, AZ (October 1988).
34. M. L. Shooman, *Software Engineering: Design, Reliability, and Management* (McGraw–Hill, 1983).
35. T. Standish, "An essay on software reuse," *IEEE Trans. Softw. Eng.*, **10** 5 (September 1984) 494–497.
36. W. Stevens, "How data flow can improve application development productivity," *IBM System Journal*, **21** 2 (1982) 162–178.
37. C. Wilson and L. Osterweil, "Omega–A data flow analysis tool for the C programming language," *IEEE Trans. Softw. Eng.*, SE–**11** 9 (1985) 832–838.
38. E. Yourdon and L. Constantine, *Structured Design* (Prentice–Hall, 1979).