UMIACS–TR–90–66 May 1990
CS–TR–2470

Data Binding Tool: a Tool for Measurement Based Ada Source Reusability and Design Assessment*

Alex Delis
Department of Computer Science
University of Maryland
College Park, MD 20742

Victor R. Basili
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

This paper reports on the *d*ata *b*inding *t*ool (*dbt*). It assists in Ada source code reusability and system design assessment. Software system components are defined in the context of the language using a flexible scheme. Data binding metrics are utilized to measure the inter-component interactions and cluster analysis is used to present structural configurations of software systems. The clustering technique and the tool design problems are discussed. Reusability potential and design assessment of examined systems are examined.

Key Words: Data Bindings, Ada, Mathematical Taxonomy, Source Reusability, Design Assessment.

*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.

1 Introduction

Software demand has exceeded industry's capacity to supply it. 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 environments, the production of software has not achieved a sufficient rate of productivity to satisfy the demand. It is expected that the trend will continue at least in the near future [Sta84].

Software reuse offers a possible solution to this problem. By reuse, we mean not only reuse of code but also of other life cycle products, as well as processes. In certain environments, reuse of processes is already a 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.

Reuse of products still remains limited. Reuse of early software life cycle products could prove extremely profitable. Good system designs could be partially re–used in future system development efforts. Reuse of source code remains an issue of critical importance.

The introduction of Ada [Def83] had two primary goals: to promote reusability of code through the wealth of the language constructs and to assist the system design through its abstraction mechanisms. The elementary building block of Ada is the subprogram structure [Boo87]. 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.

The purpose of this paper is to report on the development of an Ada based tool (Data Binding Tool or **dbt**) and demonstrate how it can assist 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 [BT75] provide a measure of component or module interaction. They evaluate the proximity of a system components. This 'closeness' is input to a mathematical taxonomy (cluster analysis) method to construct a simple, unique tree diagram of the elements involved. This diagram –usually called dendrogram– expresses element similarities and dissimilarities at a glance.

By examining dendrograms, one can determine what happens if certain components within a cluster need to be migrated into a new environment. Designers can determine if clusters represent their design

and derive a deeper understanding of the system. The newly acquired knowledge can then be used to attack design errors and promote better solutions to existing problems.

The tool has been applied to a number of software projects. Conclusions about reuse potentials were drawn. Conformance of the obtained dendrograms with the initial designs of the systems is also discussed.

The organization of the paper is as follows: In section 2, a review of related work and the concept of data bindings are presented. The notions of a 'component' and data binding for Ada are discussed in section 3. Section 4 briefly studies the concept of mathematical taxonomy and how the final output (dendrogram) is produced. In section 5, the design of the tool and the major problems encountered are examined. Sections 6 and 7 give a description of the test data and an explanation of the derived dendrograms. Finally, we conclude with a summary and status of the tool.

2 Background

2.1 Related Work

Information flow metrics and ideas –similar to the one used in this paper– have been extensively used in the literature for a broad range of goals. There are several reasons:

- They can be used in stages prior to detailed coding. Information flow metrics can be applied during PDL design.
- They provide insight into complex programming structures.
- They can be automated readily since a rather not complicated instrumentation of the compiler is usually required.

In [HK81], the concept of information flow is defined formally and the notions of fan–in and fan–out are introduced. Fan–in is the the number of flows into a procedure plus the number of data structures from which a procedure retrieves information. Fan–out is the number of flows from a procedure to its outside environment plus the number of data structures the procedure in question updates. Global and local flows are being differentiated. Global flows are those due to the existence of global data structures. Local flows are those occurring between subprograms calling each other. Based on these

concepts the authors speculate that the complexity of a procedure depends on the complexity of the code and the environment. Measurement on the suggested metrics were taken in an attempt to locate design and code problems. In the same work, information flow was used to quantify the strength of connections between program modules.

Wilson and Osterweil [WO85] used information flow to detect mistakes in C programs. Code that goes through the compilation phase successfully, may create problems in run–time. The main idea behind this work is that variables follow a sequence of events: definition, reference, undefinition. If a variable throughout the control flow of a function presents a pattern such as definition–definition, definition–undefinition, undefinition–reference then a data flow anomaly has been detected. Pointers are treated by performing reanalysis of every function at the point of its invocation. The main result of this work is that information and data flow analysis may assist substantially in detecting possible defects in the source code.

Barth [Bar78] used data flow techniques to perform interprocedural 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 P–hard. In the same paper, a one pass algorithm is presented that tries to gather complete interprocedural information. Recursive and non–recursive subprogram calls are treated differently. The emphasis of this algorithm is on the computation of 'modifies' and 'uses' information for every object (similar to the notion used in data bindings, and in [WO85]). The algorithm has been used in the construction of optimizing compilers and the generation of program diagnostics.

Information flow techniques were used in [Ste82] to advocate improvement in application development productivity. Indeed, this is the first reference which views information flow techniques in conjunction with reusability and system design assessment, as well as a mechanism for reducing complexity in systems.

Belady and Evangelisti [BE81] used the interconnection of program modules and data structures in terms of calls and references to determine system partitioning using clustering techniques. An algorithm to perform automatic clustering of modules, and a metric to qualify the complexity of the resulting module partioning, are proposed.

Hutchens and Basili [HB85] present 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 threefold : to find functional clusters, perform error analysis involving changes in the code and compare some clustering techniques.

Selby and Basili [SB88] use data bindings to quantify ratios of coupling and cohesion. They subsequently use these ratios to generate hierarchical systems descriptions to localize errors by identifying error–prone system structures during the development phase.

The work reported here is along the same lines as in [HB85] but is differentiated by two points. First the language is Ada, which is much more complicated than those languages originally used (Fortran and SIMPL). The concept of a segment and the definition of data binding need to be tailored accordingly so that they work synergistically within Ada. Second, data bindings are applied to assist in source code reusability and assess system design.

2.2 Data Bindings

Data Bindings fall in the category of measures for data visibility [BT75]. 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:

- Let α and β two program segments and variable γ global to α and β. If γ is assigned by segment α and accessed by β then there exists a data binding between these two program segments denoted by the triplet (α, γ, β).

This triplet basically describes a flow of information from the first segment to the second. It is also possible, that another binding of the reverse type of flow exists (i.e. (β, γ', α) where γ' is a global assigned by the second segment and referenced (accessed) by the first). Intra–segment bindings are not considered to be of interest since they portray flow internal to the segment (i.e. (α, x, α) does not count as an extra data binding).

In [HB85], the notion of segment was synonymous for the Fortran subroutine and function.

Several families of Data Bindings were identified in early work: potential, used, actual and control flow. The definition given above is for 'actual data bindings'. It is also the only one that has been

used so far in realistic measurement settings [BT75, HB85, SB88].

Potential bindings as the name suggests capture the 'possibility' that two segments communicate through a variable that is located in their lexical scope. 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' (that is either reference or assignment) of a variable in their scope. Thus, if α and β are two segments which 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. As the definitions become more restrictive smaller numbers of data bindings are found. Below, an example is given to demonstrate the various data binding definitions (segments are assumed to be equivalent to procedures and functions; passing of parameters is done by value).

```
procedure   EXAMPLE is
            A, B, D, E : INTEGER;
            C          : INTEGER :=0;

            procedure BAR(G : in INTEGER) is
            begin
                        C := G + B;
            end BAR;

            function COO(D, E : in INTEGER) return INTEGER is
                        TEMP : INTEGER := 0;
            begin
                        TEMP := D + E + C;
                        return (TEMP);
            end COO;

            procedure FOO(E : in INTEGER) is
            begin
                        D := 2;
                        A := COO(D, E) + C;
                        BAR(A);
            end FOO;

            procedure MAIN is
            begin
                        E := 1;
                        B := 2;
                        FOO(E);
            end MAIN;
```

```
begin
null;
end EXAMPLE;
```

According to the definition, the actual bindings found in this piece of code are:

```
(MAIN,E,FOO), (MAIN,B,BAR),   (FOO,A,BAR)
(BAR,C,COO),  (FOO,D,COO),    (BAR,C,FOO)
```

As mentioned before, potential bindings include additional triplets and their number is much higher than that of the actuals. For example, `(FOO,B,BAR)` would be a potential binding but not a used data binding. `(FOO,C,COO)` is a used data binding but not an actual data binding. Control flow bindings are less than the actuals listed above. Specifically, the triplet `(BAR,C,COO)` is not a control flow binding since there is no way control can pass from segment `BAR` to `COO`. Henceforth, data bindings are assumed to mean actual data bindings.

3 Concepts in Ada

3.1 Segments

System segments in [HB85] are called either modules or components. The term module is overloaded in the literature. There is disagreement on what a module (component) is. Myers in [Mye78] proposes that a system module (component) is a set of executable statements that should satisfy the following criteria:

- it is a closed subroutine
- it has the potential to be called from any other module in the program
- it has the potential of being independently compiled

The last two requirements are more suggestive than defining. Fortran subprograms generally comply with all the above criteria. Since, subprograms are the only constructs for abstraction in Fortran, their utilization as system components in [HB85] is justified.

Hammons in [HD84] 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

can not be called from any random point in the system (except the package body scope) they do not qualify as modules. 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 (ABC's thereafter), for defining Ada constructs as components is proposed here. ABC's 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. At the second, level certain components of the first level are viewed as integrated entities. For instance, one could view the subprograms encapsulated in a package as making up a unique and indivisible system component. On the other hand, one may view nested subprograms, not as separate components, but as parts of their incorporated construct.

ABC's of the first level are the essential building blocks of the language and of our analysis. It is interesting to note that these 'low level' components comply with Myers' laws.

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 task can be compiled separately.

A substantially distinct view of a system under analysis could be taken by recognizing there is no need for units within a package to be analyzed separately. That would change the formulation of the participating ABC's in the analysis. There are occasions where packages implement abstract data types and are required to be seen as unique (integrated) components. The same applies in 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 second level ABC's is supported by the tool. If nothing is explicitly demanded, then the analysis will be carried out assuming that only first level components are elaborated. Naturally, second level ABC's build on knowledge acquired by the analysis performed on the first level of

Ada data Binding Components.

Block statements are not considered ABC's of either level. They are rather part of the Ada Binding Components which contain them within their scope. Package body initializations are also not ABC's. Initializations are considered to be part of the package as a whole and therefore, are covered by the Ada Binding Components of the second level.

Generics offer a great utility for reusable software. Therefore, it is important to understand and classify the interaction of instantiated generics with the rest of the system. Thus, instantiated ABC's generics are identified and their data interaction with other Ada Binding Components is evaluated.

3.2 Data Bindings in Ada

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

There exists a Data Binding (abc_i,x,abc_j) between two ABC's abc_i, abc_j if and only if:

- abc_i calls abc_j.
- object x is part of the abc_j interface (formal parameter list).
- abc_i assigns to x and abc_j references it.

Note that in Ada, the mode of the formal parameter x limits the availability of the possible bindings. The binding (abc_i,x,abc_j) may exists only if x is either an *IN* or *IN OUT* type of parameter.

There exists a Data Binding (abc_j,x,abc_i) between two ABC's abc_i, abc_j if and only if:

- abc_i calls abc_j.
- object x is part of the abc_j interface (formal parameter list).
- abc_j assigns to x and abc_i references it.

The binding (abc_j,x,abc_i) exists only if x is either an *OUT* or *IN OUT* parameter.

Finally, there exists a Data Binding (abc_i,x,abc_j) between two ABC's abc_i, abc_j if and only if:

- Object's x scope extends to both abc_i and abc_j.

- abc_i assigns to x and abc_j references it.

Except for the removal of the stipulation that x be a global variable, this last definition parallels the original one [BT75]. Note that 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 clarifies exactly what the scope of each segment is. Note also that the defined bindings are those occurring through execution and not through elaboration (such as through initializations or default value assignments).

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 like 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. Input data describing the similarity of objects are summarized by the abstraction that occurs. 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; speculations and theories 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 ABC's 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 [Eve77, Har75, JRS77]. Given that programs are often organized as hierarchies of elements a bottom–up clustering algorithm appears most appropriate.
Generally, the initial 'raw' data collected on a set of n objects (having m attributes) constitutes a

matrix M with dimensions $m \times$ n.

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

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 (it says how far away every object is from all others) and has $n \times n$ dimensions. This distance matrix is passed over to an iterative algorithm that determines the objects' dendrogram.

More specifically, the clustering of Ada data Binding Components matrix M is formulated with the use of data bindings. In this case, $m = n =$ *number of ABC's* 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 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:

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

The numerator of the expression equals to the number of bindings in which either the i–th or j–th components participate but not both. The denominator expresses the number of data bindings in which either the i–th or j–th components participate. 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 occurring between the two ABC's) then $N_{i,j} = 100$. This says 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 matrix M as :

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

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 due to having the largest number of data bindings). From the initial matrix N objects b,d are those to be grouped. Then, they are fused with the nearest groups/objects (smallest distance principle). The distances between the newly created cluster *(b,d)* and the rest objects a,c,e is calculated as follows:

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

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

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

That is 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 last iteration becomes N_2 with elements *(a,c,(b,d))* and e.

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

At last, object e is fused into cluster *(a,c,(b,d))* at level 83. Graphically the created clusters are depicted as a dendrogram in figure 1. Mathematically, cluster analysis has some of 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 is that if the classification pattern for the measured objects' characteristics is reasonable, the chances that the taxonomy leads to meaningful results with respect to those characteristics is increased. In this study, the introduction of Ada Binding Components, along with the definitions of data bindings offer a clearly defined classification scheme for measurement of data use and visibility.

5 Design Outline of the dbt tool

The basic computations to be performed by the data binding tool are:

- Identification of Ada data Binding Components (ABC's).

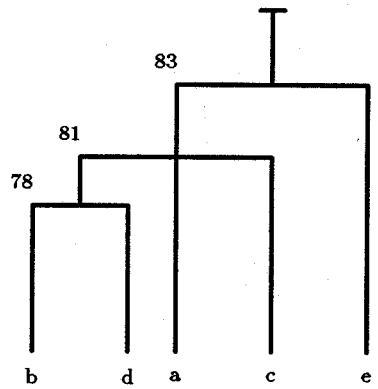

Figure 1: Sample System Dendrogram

- Computation of Data Bindings among ABC's.
- Performance of the Mathematical Taxonomy method.

The first two functions are performed simultaneously when the source of an Ada program is parsed. Their output consists of the data binding matrix M whose size is equal to the number of identified ABC's.

Lex [Les75] and Yacc [Joh75] specifications for Ada have been used to parse the input programs. Actions in the form of C [KR78] functions, were incorporated into these specifications. These actions calculate the bindings and and manipulate the matrix M. Since the definitions of bindings require information about the use of globals and the association of program entities to ABC's, a partial symbol table for the parsed program is also created.

The implementation of source code metrics using Yacc specifications alone is limited for complex structured languages such as Ada. Intermediate language representations are needed for an effective measurement process.

The main idea behind the design of the **dbt** is that the Ada source program is transformed to an 'intermediate' representation which is comprised of an interconnected set of tables and each lexical scope maintains such a table called a frame. Every subprogram, nested subprogram and package has its own frame in the structure of the intermediate representation. Frames are interconnected 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;
        };
```

name contains the name of the elaborated Ada data Binding Component. **type** points to a string that describes the kind of the frame (i.e. subprogram, package, gen_package, function etc.) **returns** is a character string which in the case of a function ABC describe the type of the object to be return. It remains null in other types of frames. **_num_id** is a unique numeric identifier for every frame in the intermediate representation and is used by the searching routines of the tool.

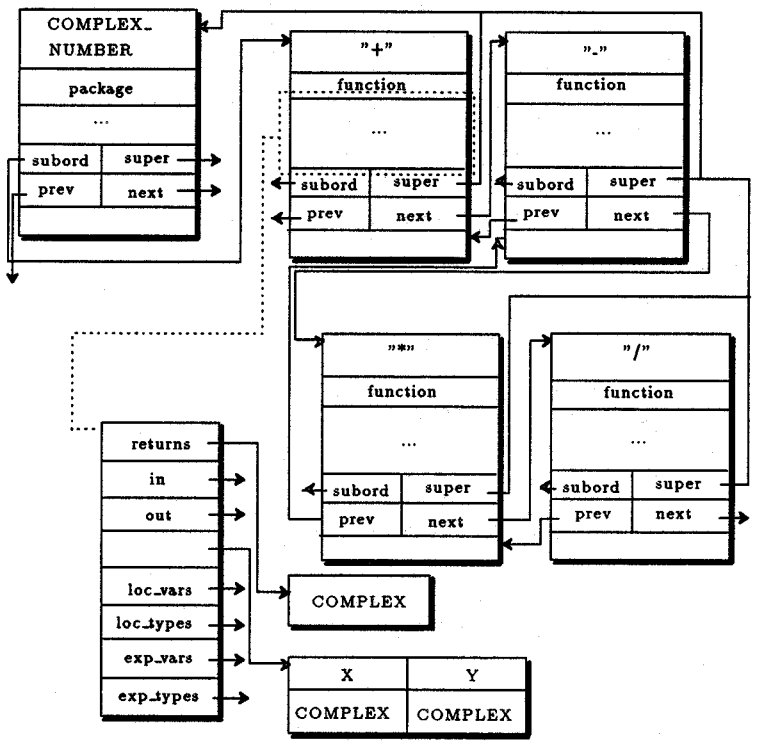

Figure 2: Intermediate Representation of an Ada package

subord, superior, next, prev are pointers that define interconnections with other structure cells. In particular, **subord** is a pointer providing access to ABC's nested in the current Component. **superior** points to the component that incorporates the current one in its scope. **next** and **prev** point to frames that belong to the same lexical level with the current frame (**next** for the next frame to be worked on and **prev** for the cell just elaborated).

_in, _out, _inout point to ABC's corresponding parameter lists. Information about local types is kept in a list pointed to by **loc_types** and those visible from the outside of the frame environment 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. **special** field is explained latter in this section. For example the intermediate representation for the package described below is shown in figure 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;
```

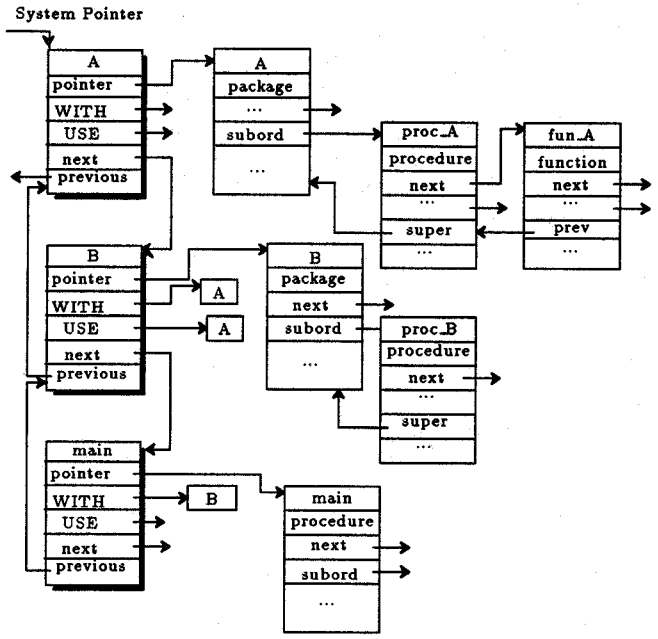

Figure 3: Top Level Structure

```
                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;
```

A reasonably sized Ada program consists of a number of packages and subprograms compiled in a predefined order to produce object code. So far, just the description of simple ABC's 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. 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 3 shows the top level structure and how visibility is maintained. This aspect of the structure permits the identification of ABC's coming from different compilation units. For instance, procedure `b_proc()` invoked in `main` establishes bindings between these two involved ABC's. 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, then 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 top level structure and identify the correct frame within a hierarchy of representation frames.

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 give a bidirectional information flow.

While parsing an ABC's body, a record of what is being either referenced or assigned is maintained. The goal of this record is to assist in bindings' computation 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.

Ada's complex structure 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 is that of subunits. A rather simple approach to overcome it would be to preprocess the source and expand the source code of the program with the subunit bodies.

Another approach 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's calls can be easily derived. On the other hand, bindings due to globals require a 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.

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 pretty much the same way. For a renamed ABC, a new frame is set up in the system representation of type 'renamed' and the field **ren_frame** points the renamed ABC (which is NULL otherwise). The cell is created within the scope 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 the instantiation, a record is kept of which ABC's are invoked in every generic Ada data Binding Component. The field **special** of the template undertakes this role.

Imported ABC frames (i.e. imported functions and subprograms) are designed to be at the same lexical level as that of generic unit in the generic structure. Figure 4 shows how frames are set up during the elaboration of the generic **FORMATTER** package. The package's outline is:

```
generic
        type WORD is private;
```

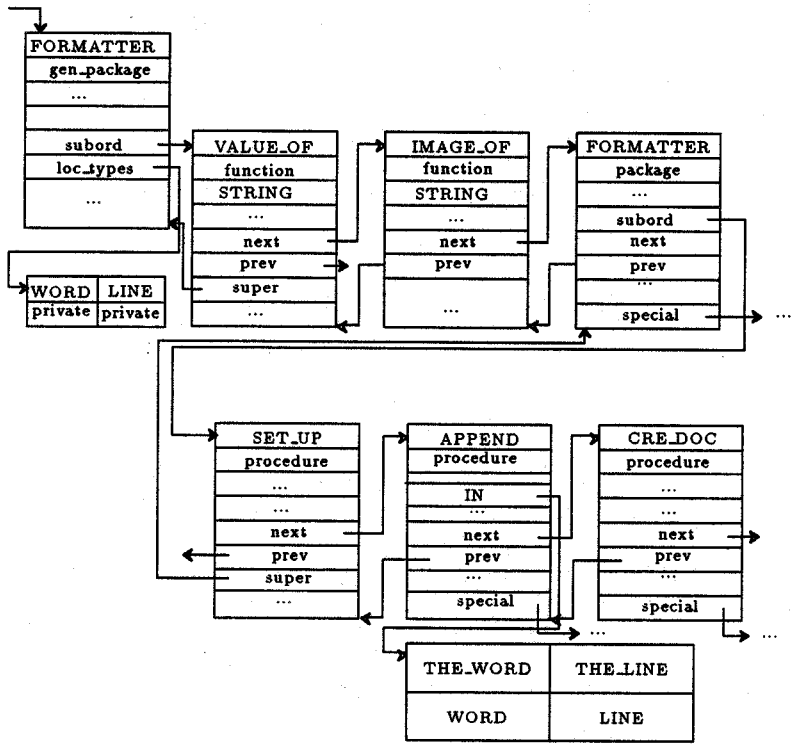

Figure 4: Frame description of a generic package

```
        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;
```

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 ABC's and the imported ABC's can be easily computed. Bindings occurring among the instantiated ABC's and the rest of the system ABC's 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 parsing and stored in the 'intermediate representation' of the Ada program is used to disambiguate the invocation of overloaded ABC's. A variation of the algorithm proposed in [Cor81] could be used to achieve resolution given the fact that the proposed representation resembles a 'decorated' tree. The current version of the tool does take some effort to disambiguate overloaded Ada Components but it is rather limited.

6 Test Data

Ada test software 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 test software contains 3 sets of Ada compilation units making up 4 systems.

Table 1 shows some of their essential characteristics such as the name of the set , name of the system, number of lines of code, number of identified low level ABC's, and provider. Every system is assigned a unique numeric identifier (number).

Most of the programs originating from the *RAPID* project were designed and implemented for reuse. Future use of these components in the organization is expected to be heavy. Some of these systems were designed using functional decomposition and the rest using variations of object–oriented design.

Programs provided by the Software Engineering Group were developed using object–oriented design. Although reusability was a concern, it can be said that it was a secondary goal for those projects.

Number	Group of Components	System	ABC's	Lines	Source
1	XDB_Ada_interface	XDB_Ada_interface	2	219	RAPID
2	ada_components	ada_components	27	1510	RAPID
3	date_formatting_pack	Date_Formatter_Package	55	3317	RAPID
4	dec_arithmetic	Decimal	61	941	RAPID
5	Registration	Register_A	17	623	*UMD*
6		Register_B	19	676	*UMD*
7	marine_corps_ _ada_comps	Calendar_Utilities	105	2260	RAPID
8		Float_Point_Utilities	116	2461	RAPID
9		String_Math	70	1420	RAPID
10	personel	Personel_Validation_ _Operations_Package	27	1583	RAPID
11	keyed_file_ _io_package	Keyed_File_ _IO_Package	12	817	RAPID
12	string_utilities	String_Utilities_ _Package	19	985	RAPID
13	subtype_con_ _utilities	Subtype_Conversion_ _Package	67	3236	RAPID
14	type_conversion_ _pack	Type_Conversion_ _Package	43	2147	RAPID
15	window_io_pack	Window_IO_Package	106	3903	RAPID
16	Electr_Mail_Sys	EMS_System	46	3059	*UMD*
17	enumeration_type_ _conversion_pack	Enumeration_Type_ _Conversion_Package	84	4454	RAPID
18	Text_Format	Text_Formatter	6	229	*UMD*
19	screen_data_ _manager_pack	Screen_and_Data _Manager_Package	79	4311	RAPID

Table 1: Ada Systems Characteristics

7 Examination of Derived Dendrograms

7.1 Classification of Dendrograms and Reusability Potential

As expected in the analysis, clustering distributes groups of Ada data Binding Components across the system. In many cases, a set of small groups (subsystems) revolve around a main cluster (system core).

At this point, an explanation of the output format of the tool, the dendogram, is given. The dendogram forms a tree structure. A component in the dendogram has three properties associated with it:

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

We will use the dendogram produced for the system `string_utilities_package` (System 12) in figure 4 for demonstration purposes. Note that ABC's appear with their fully expanded name. 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. In general, 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 only occurs at the highest level, it also means that there is no actual data binding to any other components in the system. These components are completely independent of the system although they may have potential data bindings.

At 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 cluster as figure 5 shows. At 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 at the same data are grouped in the same cluster). This creates two second level clusters containing 11 components.

The numbers at a particular pass of the clustering process are defined in a different context than the numbers at the prior pass (because of the transformation made), and so numbers do not have a consistent meaning across passes. However, there is a partial ordering based upon bindings, defined among the components, which can be calculated using the cluster numbers. Thus, `leading_non_blank_position` and `string_end_position` are more tightly bound than `trailing_nonblank_position`, `string_end_position`, and `is_empty_string`, etc.

The subtree to which a component belongs defines its clustering subsystem. For example, the `substring_position` and `character_position` cluster is grouped with the components `substitute_substring` and `substitute_character`, creating a clustering subsystem or dendogram subtree. Finally, all the clusters coalesce into one clustering subsystem, representing the full system. Figure 6 gives a pictorial view of the system clustering.

Examining each of the Ada systems from the reusability point of view, we can analyze the types of extensions as well as the potentially reusable subsystems that are available for future system development.

At the top level, each of the top level 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. 0–level components 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 deleting the functionality of a system. For example, in figure 5, the function `string_utilities_package.is_numeric_string` 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. This defines a class of easy to make extensions to the system.

```
100
    0
      string_utilities_package.fill_string
      string_utilities_package.clear_string
      string_utilities_package.next_blank_position
      string_utilities_package.is_numeric_string
      string_utilities_package.is_alphabetic_string
      string_utilities_package.change_string_case_upper
      string_utilities_package.change_character_case_upper
   75
      66
         string_utilities_package.trailing_nonblank_position
         string_utilities_package.string_length
         string_utilities_package.is_empty_string
      50
         string_utilities_package.leading_nonblank_position
         string_utilities_package.string_end_position
   80
      string_utilities_package.change_string_case_lower
      71
         string_utilities_package.string_equalities
         string_utilities_package.change_character_case_lower
      75
         string_utilities_package.substitute_substring
         string_utilities_package.substitute_character
         66
            string_utilities_package.substring_position
            string_utilities_package.character_position
```

Figure 5: A System Dendrogram

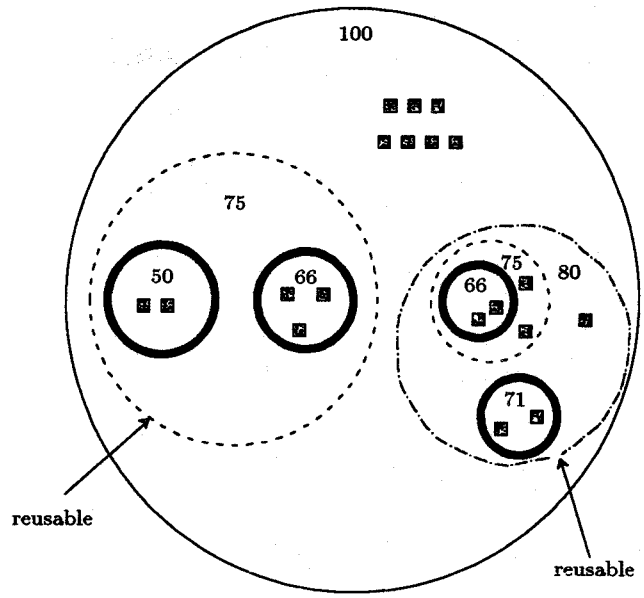

Figure 6: A View of the Clustered System

Since each of the clusters at level 75 and 80 are also independent of each other and do not use any of the ABC's from the 0 level, they themselves form reusable subsystems. Thus, a design that offers a large number of cluster at the top level provides a large set of independent components that can be used as the basis for developing other systems.

From the system dendrograms given in the appendices, 2 offer 3 reusable subsystems (e.g. the systems in Appendices 5 and 6), 4 offer 2 reusable subsystems (e.g. the systems in Appendices 1, 3 and 4, as well as figure 5) and 1 offer only 1 reusable subsystem (e.g., the system in Appendix 2).

In total, *Rapid* offers fifteen reusable systems with a total of 31 reusable subsystems. 4 *UMD* systems offer 7 reusable subsystems.

7.2 Design Assessment

Design involves a substantial effort to reduce complexity in a system under development. Good design techniques target for easily understood systems. They also enable reasonable maintenance and

enchancement efforts. Parnas work [Par72] provides perhaps the most widely accepted criterion for 'good' design, information hiding. Many design methodologies have been proposed during the last two decades around the notions of module strength and coupling. Most of this early work can be categorized as either hierarchical decomposition or bottom–up.

The essential theme in hierarchical decomposition methods is that systems are decomposed into co-operating subsystems. If needed then this decomposition process is applied as many times as it is appropriate. Issues like implementation feasibility, ability to manage the components and component complexity decide the level of nesting. Functional decomposition, top–down and stepwise refinement [Wir71] are variations of the same idea. SADT's, Structured Jackson Design, Information Flow diagrams, Structured–Composite Design fall in this category, among others.

On the other hand, bottom–up design techniques embrace the idea that primary designer's concern is the development of the elementary system units. Grouping of those units allows composition of subsystems. Hence, abstractions and virtual machines can be constructed. Use of pseudo–code, and detailed design notation are two means among others to implement this methodology. In reality, a mixture of the two methodologies is typically used.

Another technique that has received a lot of attention recently is the paradigm of object–oriented design. The main idea is that instead of trying to divide the system into components (modules) the system is structured around objects. These 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. In contrast with the previous two approaches, this methodology tends to be more 'localized' as far as changes are concerned.

Designing with one of the traditional methods should produce system with dendrograms resembling to the functional decomposition or grouping that was followed. This is true if we accept that decomposed subsystems components cooperate a lot (which is a reasonable assumption). Therefore, by looking at the dendrogram someone should be able to identify major pieces of a dendrogram corresponding to the functional decomposition. Appendix 1 presents such a dendrogram where Ada Binding Components from `date_formatter_package` (System 3) are clustered almost separately than those of `string_utilities_package`. In appendix 2, the dendrogram of system 8, `float_point_utilities` is shown. Ada Binding Components from `fixed_point_utilities`, `floating_point_utilities` and `integer_utilities` are clustered together. Another group at the same level is the one consist-

ing of `dos_utilities` and finally the last big cluster in the dendrogram contains elements from `string_utilities` and `character_utilities`.

A dendogram could be used to get an assessment of the strength and coupling of the various clusters in the system. From this point of view, we can study the dendogram to gain some insights into the design.

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 demonstrates close coupling among its components which would be equivalent to a cluster of high strength. 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 demonstrate 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 simple changes to the system might change the dendogram structure with regard to the components in this cluster.

In general, it could be said that for any level n of the dendogram, 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 dendogram is small, e.g., at most 2, independent of the depth or cluster numbers, then a highly nested structure has been encountered. If the number of siblings at any level is large, then the system has a design that was probably the result of function decomposition.

Object–oriented systems tend to built on layers. This is the point where they differ from systems designed using traditional methodologies. It should be also expected that these layers are flat, in the sense, they do not contain a great deal of language nesting. A procedure call though, may trigger a nested sequence of procedure calls throughout the system layers. Dendrograms of such systems have groups of components that carry out semantically related computation tasks through the levels of abstraction. For example in Appendix 5 computations from different levels of abstraction like `semester.addcourse` and `studentpack.addcourse` belong to the same cluster because they perform

operations on the same data. So even though `studentpack.addcourse` is at the lowest of the call structure and `semester.addcourse` is at a higher level of the call structure and yet they are both in the same cluster. Other such treegrams can be seen in Appendices 3 and 4 (Systems 5, and 7 respectively).

The program whose dendrogram is given in Appendix 3 is multi–layered and subprograms at one level are based on the operations described by the one immediately below (there are three levels in total). It is clear that there is isolation of logically cooperating components. For instance creation related routines are grouped together. The same happens with insertion and printing routines. This is also clear as well in the Appendix 4 where components from packages `character_utilities` and `string_utilities` performing logically similar operations (for example `uncapitalize`, `make_lowercase`) cluster together. To some extent in the same dendrogram separation of layers is also distinct. Components of low level implementation (i.e. `integer_utilities`) cluster at level below 40. Elements from `string_utilities` and `character_utilities` cluster mostly at level range 45–75. ABC's of `calendar_utilities` cluster at the highest level (range 70–95).

System 5 was rewritten using different internal data structures resulting in System 6. It is worth mentioning that the dendrogram for the new program (as shown in appendix 5) is substantially identical to the one obtained from the original program (appendix 3).

In conclusion, the examination of **dbt** tool produced dendrograms for the provided systems revealed insight into the design techniques used in the development process. Clusters of systems designed with object-oriented methods contain components carrying out semantically related computations. Systems designed using traditional decomposition techniques provide clusters with a similar layout to that of the system decomposition. The dendrogram also reveals the main system control thread that follows the hierarchical decomposition and is a common characteristic of systems designed with such techniques.

8 Summary and Status of the Tool

A tool for Ada source reusability and design assessment was presented in this paper. The central concept used was that of data bindings. Ada data Binding Components were defined. Data Binding metrics were also proposed for measuring Ada inter–component interactions. Mathematical taxonomy

was utilized to present structural layout of systems called dendrograms.

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 of the source code of an Ada system. The issue of intermediate program representation particularly oriented for measurement purposes need further research.

Dendrograms can be used to isolate reusable pieces of code and understand ways packages can be built on. Non–clustered code could be extracted to simplify and lessen compilation costs which can be expensive in Ada environments.

Dendrograms could be also used as an assessment tool by designers to verify the composition of their system. It is proposed that according to the type of design followed dendrograms should present certain characteristics.

A prototype of the **dbt** has been implemented. Ada data Binding Components of both levels are identified and all sequential aspects of the language have been developed. Appendix 6 shows the dendrogram of System 6 for the lower level abstraction routines of package `courselistpack` integrated into a second level ABC. The extension to the concurrent elements of the language is straightforward. The tool is limited with respect to dealing with overloading but serves as a beneficial prototype. The tool is being integrated into the **TAME** [BR88] and **CARE** [BC88] systems under development at the University of Maryland.

9 Aknowledgements

The authors wish to acknowledge helpful discussions with H. Dieter Rombach, Gianluigi Cardiera and especially John Bailey for his comments on an earlier version of this paper that helped us to improve the presentation.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers:principles, techniques and tools.* Addison–Wesley, 1986.

[AWW88] L. Clarke A. Wolf and J. Wileden. A Model of Visibility Control. *IEEE Transactions on Software Engineering*, SE–14(4), 1988.

[Bar78] J. Barth. A Practical Interprocedural Data Flow Analysis Algorithm. *Communications of ACM*, 21(9), September 1978.

[BC88] V. Basili and G. Caldiera. Reusing Existing Software. Technical report, Dept. of Computer Science, Un. of Maryland–CP, October 1988.

[BE81] L. Belady and C. Evangelisti. System Partition and its Measure. *The Journal of Systems amd Software*, February 1981.

[Boo87] G. Booch. *Software Engineering using Ada.* Benjamin–Cummings, second edition, 1987.

[BR88] V. Basili and D. Rombach. THE TAME PROJECT:Towards Improvement–Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[BRBD88] V. Basili, D. Rombach, J. Bailey, and A. Delis. Ada Reusability Analysis and Measurement. In *Proceedings of the 6th Symposium on Empirical Foundations of Information and Software Sciences*, October 1988.

[BT75] V. Basili and A. Turner. Iterative enchancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, 1(1):390–196, December 1975.

[CL83] R. Cook and T. Leblanc. A Symbol Table Abstraction to implement Languages with Explicit Scope Control. *IEEE Transactions on Software Engineering*, SE–9(1), January 1983.

[Cor81] G. Cormack. An Algorithm for the selection of overloaded functions in Ada. *SIGPLAN Notices*, 16(2):48–52, 1981.

[Def83] United States Department Of Defense. *Reference Manual for the Ada Programming Language*, ansi–mil–std–1815a–1983 edition, February 1983.

[Eve77] B. Everitt. *Cluster Analysis.* Heinemann Educational Books, London, UK, 1977.

[Har75] J. Hartigan. *Clustering Algorithms.* John Wiley and Sons, New York, NY, 1975.

[HB85] D. Huntchens and V. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Soft. Engineering*, 11(8), August 1985.

[HD84] C. Hammons and P. Dobbs. Coupling, Cohesion, and Package unity in Ada. *ACM Ada Letters*, IV(6):49–59, 1984.

[HK81] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions On Software Engineering*, SE–7(5), September 1981.

[Joh75] S. Johnson. Yacc–yet another compiler compiler. Technical report, AT&T Bell Laboratories, 1975.

[JRS77] Nicholas Jardine and Robin Sibson. *Mathematical Taxonomy.* John Wiley and Sons Ltd., May 1977.

[KR78] B. Kernigham and D. Ritchie. *The C Programming Language.* Prentice–Hall Software Series, 1978.

[Les75] M. Lesk. Lex–a lexical analyzer generator. Technical report, AT&T Bell Laboratories, 1975.

[Mye78] G. Myers. *Composite–Structured Design.* Van Nostrand Reinhold Company, 1978.

[Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of ACM*, December 1972.

[SB88] R. Selby and V. Basili. Error localization during software maintenance: generating hierarchical system descriptions from source code alone. In *Proceedings of the Conference on Software Maintenance*, Phoenix,AZ, October 1988.

[Sta84] T. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, 10(5), September 1984.

[Ste82] W. Stevens. How data flow can improve application development productivity. *IBM System Journal*, 21(2), 1982.

[Wir71] N. Wirth. Program development by Stepwise Refinement. *Communications of ACM*, April 1971.

[WO85] C. Wilson and L. Osterweil. Omega–A Data Flow Analysis Tool for the C Programming Language. *IEEE Transactions on Software Engineering*, SE–11(9), 1985.

Appendix 1

```
 Dendrogram:
100
    0
      date_formatter_package.valid_date date_formatter_package.valid_date
      date_formatter_package.valid_date string_utilities_package.fill_string
      string_utilities_package.clear_string string_utilities_package.next_blank_position
      string_utilities_package.is_numeric_string string_utilities_package.is_alphabetic_string
   80
      string_utilities_package.change_string_case_lower
      75
         string_utilities_package.substitute_substring string_utilities_package.substitute_character
         66
            string_utilities_package.substring_position string_utilities_package.character_position
      71
         string_utilities_package.string_equalities string_utilities_package.change_character_case_lower
    0
      string_utilities_package.change_string_case_upper string_utilities_package.change_character_case_upper
   96
      80
         75
            string_utilities_package.string_end_position string_utilities_package.leading_nonblank_position
         66
            string_utilities_package.trailing_nonblank_position string_utilities_package.string_length
            string_utilities_package.is_empty_string
      95
         date_formatter_package.valid_date
         date_formatter_package.numeric_date_string_to_full_date_string
         date_formatter_package.numeric_date_string_to_abbreviation_date_string
         92
            date_formatter_package.full_month_position
            91
               90
                  date_formatter_package.is_leap_year
                  85
                     83
                        date_formatter_package.date_record_to_abbreviation_date_string
                        date_formatter_package.create_year_string
                        date_formatter_package.date_record_to_full_date_string
                     84
                        date_formatter_package.create_two_digit_string
                        date_formatter_package.numeric_date_to_numeric_date_string
                        date_formatter_package.date_record_to_numeric_date_string
                     84
                        date_formatter_package.numeric_date_to_full_date_string
                        date_formatter_package.numeric_date_to_abbreviation_date_string
                        82
                           date_formatter_package.get_day_and_month_from_numeric_date
                           date_formatter_package.numeric_date_to_date_record
               89
                  87
                     84
                        date_formatter_package.full_date_string_to_numeric_date
                        83
                           date_formatter_package.full_date_string_to_numeric_date_string
                           date_formatter_package.get_string_segment_bounds
                           date_formatter_package.full_date_string_to_abbreviation_date_string
                           date_formatter_package.full_date_string_to_date_record
```

```
            81
               date_formatter_package.abbreviation_date_string_to_numeric_date
               date_formatter_package.date_record_to_numeric_date
               80
                  date_formatter_package.get_numeric_date_value
                  date_formatter_package.numeric_date_string_to_numeric_date
            85
               date_formatter_package.abbreviation_date_string_to_numeric_date_string
               83
                  date_formatter_package.abbreviation_date_string_to_full_date_string
                  date_formatter_package.abbreviation_month_position
                  date_formatter_package.abbreviation_date_string_to_date_record
         88
            date_formatter_package.set_current_year_string_and_valid_year_range
            date_formatter_package.numeric_date_string_to_date_record
            77
               date_formatter_package.is_numeric_string
               64
                  date_formatter_package.valid_date
                  date_formatter_package.is_day_in_month
```

Appendix 2

```
 Dendrogram:
100
   91
      string_utilities.strip_leading
      89
         50
            character_utilities.is_control string_utilities.is_control
         88
            string_utilities.right_justify
            85
               string_math.is_whole_number
               80
                  string_utilities.embedded_space
                  75
                     string_math.is_comma_separated string_utilities.scan_for
            87
               string_utilities.allow_other_characters
               85
                  character_utilities.is_equal
                  82
                     string_utilities.location_of string_math.place_commas string_math.hundreds_place
               83
                  string_utilities.is_equal
                  80
                     string_utilities.number_of string_utilities.disallow_other_characters
               86
                  string_utilities.replace
                  83
                     string_math.place_decimal_point
                     78
                        string_math.add
                        76
                           string_math.multiply string_math.make_number
                  85
                     string_math.remove_decimal_point string_math.remove_commas
                     81
                        string_math.remove_dollar_sign string_utilities.strip
            80
               string_utilities.is_null
               66
```

```
            character_utilities.is_null string_utilities.left_justify
      50
         character_utilities.is_graphic string_utilities.is_graphic
      50
         character_utilities.is_alphanumeric string_utilities.is_alphanumeric
      88
         85
            80
               string_utilities.make_lowercase
               60
                  string_utilities.uncapitalize character_utilities.make_lowercase
            83
               string_utilities.is_alphabetic character_utilities.is_alphabetic
               75
                  character_utilities.lowercase
                  66
                     character_utilities.is_uppercase string_utilities.is_uppercase
         87
            85
               character_utilities.uppercase
               60
                  string_utilities.is_less_than character_utilities.is_less_than
               60
                  string_utilities.is_greater_than character_utilities.is_greater_than
            83
               80
                  75
                     string_utilities.is_digit character_utilities.is_digit
                  75
                     string_utilities.is_special character_utilities.is_special
               75
                  string_utilities.is_lowercase character_utilities.is_lowercase
            80
               string_utilities.make_uppercase
               60
                  string_utilities.capitalize character_utilities.make_uppercase
   0
      string_math.divide string_math.subtract string_math.place_dollar_sign
      string_utilities.allow_only_characters string_utilities.number_of
      string_utilities.replace string_utilities.uncapitalize string_utilities.capitalize
      string_utilities.make_lowercase string_utilities.make_uppercase string_utilities.centered
      character_utilities.lowercase_of character_utilities.uppercase_of character_utilities.index_of
      character_utilities.image_of character_utilities.value_of
```

Appendix 3

```
 Dendrogram:
100
   studentpack.name
   50
      semester.print studentpack.print courselistpack.print
   85
      75
         semester.addcourse studentpack.print_name
         70
            semester.print_name register
         50
            studentpack.addcourse courselistpack.insert
      81
         studentpack.id
         69
            semester.dropcourse
            50
```

```
            studentpack.dropcourse courselistpack.delete
      75
         courselistpack.create
         57
            studentpack.create semester.enroll
```

Appendix 4

```
 Dendrogram:
100
     0
       calendar_utilities.short_month_name_is calendar_utilities.long_month_name_is
       calendar_utilities.current_system_date_and_time calendar_utilities.value_of
       calendar_utilities.interval_of calendar_utilities.duration_of
       calendar_utilities.value_of calendar_utilities.time_of
       calendar_utilities.time_of calendar_utilities.period_of calendar_utilities.month_of
       calendar_utilities.month_of integer_utilities.value_of integer_utilities.is_even
       integer_utilities.is_odd integer_utilities.is_zero integer_utilities.is_negative
       integer_utilities.is_natural integer_utilities.is_positive integer_utilities.max
       integer_utilities.max integer_utilities.min integer_utilities.min
       string_utilities.allow_only_characters string_utilities.number_of
       string_utilities.replace string_utilities.uncapitalize string_utilities.capitalize
       string_utilities.make_lowercase string_utilities.make_uppercase
       string_utilities.centered character_utilities.lowercase_of
       character_utilities.uppercase_of character_utilities.index_of character_utilities.value_of
     0
       calendar_utilities.current_system_date calendar_utilities.date_image_of
    60
       character_utilities.image_of
       40
          integer_utilities.image_of integer_utilities.based_image
    93
       92
          75
             string_utilities.is_equal
             66
                string_utilities.number_of string_utilities.disallow_other_characters
          91
             string_utilities.strip
             89
                88
                   85
                      80
                         string_utilities.make_lowercase
                         60
                            string_utilities.uncapitalize character_utilities.make_lowercase
                      83
                         character_utilities.is_alphabetic string_utilities.is_alphabetic
                         75
                            character_utilities.lowercase
                            66
                               character_utilities.is_uppercase string_utilities.is_uppercase
                   87
                      83
                         75
                            character_utilities.is_lowercase string_utilities.is_lowercase
                         75
                            character_utilities.is_digit character_utilities.is_special
                            string_utilities.is_digit string_utilities.is_special
                      80
                         string_utilities.make_uppercase
                         60
                            string_utilities.capitalize character_utilities.make_uppercase
```

```
                85
                   character_utilities.uppercase
                   60
                      string_utilities.is_greater_than character_utilities.is_greater_than
                   60
                      string_utilities.is_less_than character_utilities.is_less_than
            85
               82
                string_utilities.location_of character_utilities.is_equal
               83
                string_utilities.allow_other_characters
                77
                   string_utilities.replace
                   69
                      calendar_utilities.image_of string_utilities.strip_leading
            50
               string_utilities.is_control character_utilities.is_control
            50
               string_utilities.is_graphic character_utilities.is_graphic
            50
               string_utilities.is_alphanumeric character_utilities.is_alphanumeric
         90
            89
            string_utilities.right_justify
            66
               string_utilities.scan_for string_utilities.embedded_space
            57
               calendar_utilities.day_of calendar_utilities.julian_date
            57
               calendar_utilities.time_image_of calendar_utilities.current_system_time
            88
               81
                calendar_utilities.month_number_is calendar_utilities.day_part_is
                calendar_utilities.year_part_is
               50
                calendar_utilities.days_in calendar_utilities.is_leap_year
            76
               calendar_utilities.julian_part_is
               66
                calendar_utilities.days_in calendar_utilities.day_of
                calendar_utilities.time_of
            83
               66
               character_utilities.is_null string_utilities.left_justify
            75
               string_utilities.is_null calendar_utilities.convert_date
      85
         calendar_utilities.gt calendar_utilities.valid_julian_date
         calendar_utilities.lt calendar_utilities.ge
         calendar_utilities.le calendar_utilities.ne calendar_utilities.eq
```

Appendix 5

```
 Dendrogram:
100
    0
      semester.enroll studentpack.name studentpack.create courselistpack.delete
      courselistpack.create
   75
      studentpack.print_name
      25
         semester.print_name register
   75
```

```
      studentpack.id
      33
         semester.addcourse studentpack.addcourse
      33
         semester.dropcourse studentpack.dropcourse
   75
      71
         60
            courselistpack.empty courselistpack.print
         40
            courselistpack.insert courselistpack.add
      50
         semester.print studentpack.print
```

Appendix 6

```
 Dendrogram:
100
   50
      courselistpack studentpack.print semester.print
    0
      semester.enroll studentpack.name studentpack.create
   75
      studentpack.id
      33
         semester.dropcourse studentpack.dropcourse
      33
         semester.addcourse studentpack.addcourse
   75
      studentpack.print_name
      25
         semester.print_name register
```