UMIACS–TR–90–73
CS–TR–2478

May 1990

Ada Reusability and Measurement*

V.R. Basili, H.D. Rombach, J. Bailey, and A. Delis
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

The demand for software has exceeded the industry's capacity to supply it. Although advances in software development technology have increased the efficiency of developers, none have provided the dramatic improvements in quality and productivity which will be necessary to meet the current and future demands. Software reuse provides an answer to this dilemma. This paper describes two reuse studies performed at the University of Maryland Department of Computer Science. The first study defines a means of measuring data bindings to characterize and identify reusable (sets of) components of existing software. The second study defines ideally reusable components and a way of measuring the distance from that ideal for any given component. One important result of both studies has been the identification of a set of guidelines which can be used to assist developers to create more reusable software, to select reusable components from existing software, and to modify existing software to improve its generality and reusability while preserving its functionality.

*Research for this study was supported by Airmics grant 19K-CN983-C to the University of Maryland. An earlier version of this paper has been presented at the 6th Symposium on Empirical Foundations of Information and Software Sciences, Atlanta, GA, October 19-21, 1988.

ADA REUSABILITY AND MEASUREMENT

V. R. Basili, H. D. Rombach,
J. Bailey, and A. Delis

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

0. Introduction

The demand for software has exceeded the industry's capacity to supply it. Projects are frequently scaled down, delayed or even cancelled because of the time and effort required to develop the software for them. Further, the demand for software will continue to increase in the forseeable future. Although advances in software development technology such as high level languages, interactive development environments, and formal methods, have increased the efficiency of developers, none have provided the dramatic improvements in productivity which will be necessary to meet the current and future demands.

Software reuse provides an answer to this dilemma. Broadly defined, software reuse includes more than the repeated use of particular code modules. Other products such as specifications or test plans can be reused, software development processes such as verification techniques or cost modeling methods are reusable, and even intangible products such as ideas and experience contribute to the total picture of reuse. Although process and tool reuse is common practice, lifecycle product reuse is still in its infancy. Ultimately, reuse of early lifecycle products might provide the largest payoff, however for the near term, gains can be realized and further work can be guided by understanding how software can be developed with a minimum of newly-generated source lines of code.

This paper describes several parallel studies being conducted at the University of Maryland Department of Computer Science which address

various related software reuse topics. Two of these studies are covered in detail. One defines a means of measuring data bindings to characterize and identify various reusable subsets of existing software. Cluster analyses are performed to analyze the locations and strengths of data bindings, resulting in a kind of contour map showing the data binding strengths in the analyzed software. The other study defines theoretically highly reusable software and a way of measuring distances from that ideal. The measure is based on the amount of transformation required to modify existing code so that it approaches that ideal. The extent of the transformations which are needed constitutes a useful indication of the effort to reuse a body of software. These two efforts support each other, since the identification of data bindings contributes to an understanding of the transformations required to maximize the independence of a unit of software.

One important result of these efforts has been the identification of a set of guidelines which can be used to assist developers to create more inherently reusable software, to select reusable parts from existing software, and to modify existing software to improve its generality and reusability while preserving its functionality. The guidelines which were derived from each of the studies have been summarized at the end of their respective sections. Although they are written with respect to the development and reuse of systems written in the Ada language, since Ada is the medium for these studies, they apply generally to software engineering.

1. Related Reuse Projects

The two studies introduced above are part of a set of reuse-related projects being conducted at the University of Maryland Department of Computer Science. One of the other studies forms a foundation for the rest of the projects by providing a scheme to describe and classify reuse research according to several possible dimensions. The following introduces two of the dimensions of that classification scheme. These are then used to show how the other projects can be given both context and scope in terms of their contribution to issues of reuse.

Beginning with the distinction between software processes and software products as an organizational dimension, the study of reuse can be described as the study of the creation and the use (processes) of reusable software products. Although attempting to study just the processes or the product alone necessarily involves at least some knowledge and assumptions about the other, it is important to define at least the central focus for a given research effort. For example, the data binding and transformation measuring projects, the principal studies in this paper, emphasize an understanding of what constitutes a viable reusable product. They do not emphasize reuse processes, such as that of retrieving a reusable product when one is needed.

Another dimension which is useful for scoping a reuse research effort is whether the work is more concerned with product syntax or semantics. For example, the two principal projects in this paper do not address the practical issues of whether a reusable product is actually useful, but rather are limited to an examination of the structure of a product in order to assess whether it is theoretically usable in different contexts. As with the process-product division, there are interactions between syntactic and semantic research. For example, a reusable product which is very general and flexible, a syntactic aspect, may not encapsulate a sufficient amount of functionality to make its use worthwhile, a semantic issue. Nevertheless, it is necessary to develop our knowledge in one direction at a time to avoid being overwhelmed by the total number of directions that are available. For further details, this categorization of the dimensions of software reusability has been outlined in [1].

In addition to this conceptual work to develop a reuse classification scheme, there are five other reuse-oriented projects at the Department. The second project is a model of the flows of reusable information in the form of both processes and products which occur in a software development organization [2]. The third and fourth projects are general tool developments, including an Ada Static Source Code Analyzer which accepts syntactically correct Ada and performs various counts pertaining to the usage of Ada language constructs, facilities, and capabilities of that code [3], and an Ada Test Coverage Analyzer which instruments syntactically correct Ada so that test coverage can be computed at run time. These two are not strictly reuse-related

studies, however they contribute to needs that arise during the selection, evaluation, and reuse of existing products and processes. The final two projects, the Ada data binding analyzer and the transformation-for-reusability technique introduced above, are partly conceptual but are currently revealing guidelines to enhance reuse. These guidelines, along with the need to automate the analyses, will evolve into specifications for specific reuse-oriented tools. It is these two projects that are covered in the remainder of this paper.

2. Ada Data Binding Analysis

The first of the two projects which are covered in detail in this paper is a data binding analysis technique for Ada. This research seeks to evaluate the reusability of Ada software systems by analyzing the interconnectivity among their components. An Ada Data Binding Analyzer has been designed whose function is to accept syntactically correct Ada source code as input and compute various data binding measures. Data binding measures are used for characterizing the inter-module structure (interfaces) of Ada programs, where any active program unit (non-package, non-generic) can be considered a module. In addition, cluster analysis is performed to group modules of a system on the basis of the strength of their coupling. The research involves developing a technique to compute and weight coupling strengths, where coupling is based on references to variables and parameters (data binding). It also involves the interpretation and validation of those results.

The definition of a data binding given in [4] is:

> Let x be a global variable and p and q program components. If p assigns x and q references it, then the triple (p, x, q) is a data binding between the two program segments.

The existence of the data binding triple may mean that q is dependent on the performance of p because of the global x. Clearly, the order of the three elements is important since binding (q, x, p) is not identical to (p, x, q) though it can be the case that both bindings are present. The triple identifies a unidirectional communication path between the two modules. The total number of bindings among the system modules represents the degree of connectivity among the component pairs

within a system structure.

The above definition describes a particular form of binding, known as an Actual Data Binding according to the classification of four progressively stronger bindings which are given in [5] and paraphrased here:

1. A Potential Data Binding (PDB) is defined as a triple *(mod1, x, mod2)* where *mod1* and *mod2* are modules which are both in the static scope of the variable *x*. A PDB reflects the potential of a data interaction between the two components based only upon the locations of *mod1, mod2* and *x*.

2. A Used Data Binding (UDB) is defined as a PDB where both *mod1* and *mod2* make use of *x* for either reference or assignment. It reflects an active relationship between *mod1* and *mod2*. Generally speaking, UDB is harder to calculate than PDB since the implementation of components must be analyzed to discover the references or assignments to *x*.

3. An Actual Data Binding (ADB) is defined as a UDB where *mod1* assigns a value to *x* and *mod2* references it. The ADB relation is not commutative like the previous two since it distinguishes between reading the value of *x* and updating it. Because of this distinction, the calculation of ADB is more complex that that of UDB. An ADB indicates that there could be a flow of information from *mod1* to *mod2* via the variable *x*. The relative order of the execution of usages of *x* in the different modules is ignored, however.

4. A Control Flow Data Binding (CFDB) is defined as an ADB where control can pass to *mod2* after *mod1* has had control. To distinguish between an ADB and a CFDB, the possible control flows through the program must be analyzed to decide that the value provided by *mod1* could be referenced by *mod2* at some time before the program terminates. Because of the added control analysis, recognizing CFDB's is substantially more difficult than only recognizing ADB's.

Actual Data Bindings were first applied in the SIMPL family of languages and described in [4] to examine issues of visibility. In [5], Actual Data Bindings were utilized to determine the modularity of FORTRAN programs. However, these definitions need to be extended for an Ada environment where the role of global variables can be minimized through the use of local scoping and persistent but hidden variables.

By extending the definition to include parameters as well as variables, a more reasonable representation of data bindings in Ada code is possible. Such a view of Ada software can provide useful information about the viability of reusing various substructures within that code.

In theory, there should be some way of characterizing the production and consumption of data values in a program which can provide guidance about where reusable code in that program might be found. For example, in a well-structured Ada program that lends itself well to the reuse of its components, we might expect to find that references to any given variable are localized (as would occur with state variables or data structures in a package body) while references to a parameter of a subprogram or entry are more widely distributed (as in the case of an abstract data type).

Note that the analyses does not deal with problems of aliasing. Currently, only one level of data bindings is examined. Resolving aliasing would complicate the analysis task significantly, requiring data flow analysis techniques, and has not been attempted at this time.

In addition to the inclusion of parameters along with global variables, the definition of a global variable needs to be extended for Ada to mean not only variables with library-level scopes but also any variable whose scope includes both of the modules under consideration. This could be a variable which is local to one module but global to another (where the second module is nested within the first) or a variable which is external to both modules but is not necessarily at the library level.

Moreover, the definition of a module must be determined when analyzing Ada code for data bindings. For this, the Myers definition of a module was used as a point of departure. In [6], a module is a set of executable program statements that meet three criteria:

a. It is a closed subroutine; it implements a distinct piece of computation
b. It can be called from any other module in the program
c. It can be compiled separately

Based on Myers' characteristics, subprograms (procedures and functions) qualify as modules. Ada subprograms are constructs of a "closed subroutine" nature, and with minor restrictions can be separately compiled. Further, library unit subprograms as well as subprograms declared in the visible part of a library unit package can

be called from any other module in the system, assuming the use of an appropriate context clause. By extending the definition to enclosed scopes, any subprogram, library level or nested, qualifies as a module. No differentiation is made between the declaration and the body of a subprogram, even if they are in different compilation units. Both are considered to be part of the same module.

Tasks are also considered modules for the same reasons. Although the calls to a task are actually calls to entries, similar to calls to the visible operations of a package, the task itself is active, unlike a package. Again, a task declaration and body are together considered a single module. Unlike subprograms and tasks, packages are not active and cannot be called, so they are not considered modules. Rather, they constitute a (possibly empty) set of modules.

When performing data binding analyses for the purpose of examining reusability, generics are not considered separately, but follow the same classification as their non-generic counterparts. Therefore, only generic subprograms are considered, and the generic parts (generic formal parameter lists) are disregarded in the analysis. It would also be possible to examine the bindings between generic instantiations and other modules, however the instantiations of a generic are not particularly interesting when the goal is to discover reusable portions of the analyzed software. Other possible candidates for modules which were rejected include block statements, package body initializations, and accept statements. It would be theoretically possible to include these structures as modules, however they did not fit easily into the chosen definition by Myers.

The following are two definitions which are specific to Ada and which arise from the preceding discussion.

Definition I: Let *mod1* and *mod2* be two subprograms or tasks where *mod1* calls *mod2*, and let the object *x* be a part of the *mod2* interface (formal parameter list), then if *mod1* assigns *x* and *mod2* references it the binding *(mod1, x, mod2)* exists, and if *mod2* assigns *x* and *mod1* references it the binding *(mod2, x, mod1)* exists.

Note that in Ada the mode of the formal parameter *x* limits the availability of the possible bindings. The binding *(mod1, x, mod2)* can

only exist if *x* is an **in** or **in out** parameter and the binding *(mod2, x, mod1)* can only exist if *x* is an **in out** or **out** parameter. The mode does not guarantee a binding, however, since both an assignment and a reference must also be made.

> Definition II: Let *mod1* and *mod2* be two subprograms or tasks and let the the scope of object *x* extend to both *mod1* and *mod2* . If *mod1* assigns *x* and *mod2* references it, then the binding *(mod1, x, mod2)* exists.

Except for the removal of the stipulation that x be a global variable, this second definition parallels the original definition in [4]. Note that these definitions can be applied whether or not the modules in question are visible at the library level or nested, or even if they are nested inside one another. Note also that the defined bindings are those that occur through execution and not through elaboration (such as through initializations or default value assignments).

2.1 Cluster Analysis

After the set of data bindings in a program has been identified, a cluster analysis is performed to identify which modules are strongly coupled with other modules and therefore may not be good candidates for reuse, and which modules are found to be independent of others and therefore potentially useful on their own. A cluster analysis results in a hierarchical system decomposition based on the strength of the data bindings among the modules.

To perform a cluster analysis on a set of modules (typically a complete program), first a matrix, **B**, is constructed that is of $N \times N$ dimensions, where N is the number of components. Each matrix element $\mathbf{B}(k,l)$ contains the total number of bindings between component k and component l. (The direction of the binding is no longer important, so the matrix is symmetric with a zero diagonal.)

The algorithm used to identify clusters is then applied iteratively in a bottom up fashion. The first clusters are identified as those components that are bound with the highest strength, according to the matrix. A cluster is then redefined as a single module, and a new

(smaller) matrix is computed. The process is repeated, each time collapsing modules into single modules for the next iteration, building a tree-gram of the system modules. Eventually, all the elements coalesce in a single group that is the complete system cluster.

The iteration process is documented in the form of a tree-gram that expresses the differences and similarities (with respect to data bindings) of the components involved in the analysis. The specific cluster analysis technique used in this project is more completely described in [5].

2.2. Reuse Guidelines Based on Data Bindings

The data binding metrics and cluster analysis techniques introduced above have been manually applied to a limited set of small Ada programs. Based upon this experience a set of guidelines has been derived for developers to keep in mind when designing and building reusable Ada components:

- Avoid multiple level nesting in any of the language constructs.

Multiple level nesting of components results in a deeply clustered tree-gram from which it is difficult to extract reuse candidates. The use of a single level of components results in very flat tree-grams and, therefore, reusable components can be extracted much more easily.

- The use of the "use" clause is not recommended.

Data binding analysis becomes more complicated in the presence of "use" clauses since naming and visibility mechanisms must be employed in order to decide upon the correct binding of names. Further, the readability of programs is increased when expanded names are used.

- The interfaces of the subprograms should use the appropriate abstraction for the parameters passed in and out.

The abstraction of the interface components should be at the appropriate level. In this way a large number of parameters in the

formal parameter list which could make the analysis complicated can be avoided.

• Components should not interact with their outer environment.

The software should not deal with globals, should be free of side-effects and should reference objects at a local level.

• Appropriate use of packaging could greatly accommodate reusability.

Packaging is a logical way to group homogeneous collections of objects, subprograms and tasks. Even though subprograms from the same package sometimes do not cluster together due to the nature of the implemented concept (abstract data types often exhibit this tendency since they often share one or more type definitions but not variables) it is still true that reasonable packaging can assist enormously in the effort to reuse software.

3. Reuse Through Generalization: Measurement and Transformation

This project seeks to define transformation techniques which can be applied to existing Ada software in order to extract reusable functionality from it. By measuring the amount of transformation which must be performed to convert an existing program into one composed of maximally reusable components, an indication of the reusability of that program can be obtained. After applying all of the transformations which are reasonable or practical, if there remain desired transformations that cannot be performed cost effectively, then those unapplied transformations constitute a measure of the latent non-reusability of the software.

The advantages of transforming existing software into reusable components, rather than creating reusable components as an independent activity, include: 1) software development organizations are likely to have a large supply of previous projects which could yield reusable components, 2) developers of ongoing projects do not need to adjust to new and possibly unproven methods in an attempt to develop reusable components, so no risk or development overhead is introduced, 3)

transformation work can be accomplished in parallel with line developments but be separately funded (this is particularly applicable when software is being developed for an outside customer who may not be willing to sustain the additional costs and risks of developing reusable code), 4) the resulting components are guaranteed to be relevant to the application area and recognizable to the developers, and 5) the cost is low and controllable.

As discussed in the first section, this project only attempts to identify theoretically reusable software. Thus, it is concerned only with the syntax of reusable software. It does not address issues of practical reusability, such as whether a reusable component is useful enough to encourage other developers to reuse it instead of redeveloping its function. The goal of the transformations is to identify and extract components from a program which are not dependent on external declarations, information, or other knowledge. Transformations are needed to derive such components from existing software systems since inter-component dependencies arise naturally from the customary design decomposition and implementation integration processes used for software development.

To guide the transformations which will result in increased independence among the components of a program, a model is used which distinguishes between software function and the declarations on which that function is performed. The functions become the independently reusable parts and the declarations become the application-specific parts of the software. However, this is not to say that only strict functions are extracted for reuse since the model is recursive. A declaration (also known as an object) can be seen as being constructed from lower level declarations onto which some functionality has been grafted. Similarly, this new declaration can be combined with other declarations and functionality to yield yet another declaration which solves an even larger portion of the overall problem being solved. This model is somewhat analogous to the one used in Smalltalk programs where objects are assembled from other objects plus programmer-supplied specifics. However, it is meant to apply more generally to Ada and other languages that do not have support for dynamic binding and full inheritance, features that are in general unavailable when strong static type checking is required.

Applying this model to existing software means that any lines of code which represent reusable functionality must be encapsulated and parameterized in order to make them independent from their surrounding declaration space (if they are not already independent). The language construct in Ada which enables this transformation is, of course, the generic. Generics that are derived by generalizing existing program units, through the removal of their dependence on external declarations, can then be offered as independently reusable components for other applications.

Unfortunately, declarative independence is just one way that a program unit can rely on its external environment. Removing the compiler-detectable declarative dependencies and producing a new generic unit is no guarantee that the new unit will actually be independent. There can be dependencies on data values that are related to values in neighboring software, or even dependencies on protocols of operation that are followed at the point where a resource was originally used but which might not be followed at a point of later reuse. (An example of these kinds of dependencies is given later in this section.) To be completely useful, the transformation process would need to identify and remove this other dependence as well as declarative dependence. For now, however, this work only acknowledges these additional dependencies while concentrating on mechanisms to measure and remove declarative dependence.

3.1 Declarative Dependence

In a language with strong static type checking, such as Ada, any information exchanged between communicating program units must be of some type which is available in both units. Since Ada enforces name equivalence of types, where a type name and not just the underlying structure of a type introduces a new and distinct type, the declaration of the type used to pass information between units must be global to both of those units. The user of a resource, therefore, is constrained to be in the scope of all type declarations used in the interface of that resource. In a language with a fixed set of types this is not a problem since all possible types will be globally available to both the

resource and its users. However, in a language which allows user-declared types, any inter-module communication with such types must be performed in the scope of those programmer-defined declarations. This means that the coupling between two communicating units increases from data coupling to external coupling (or from level two to level five on the traditional seven-point scale of Myers, where level one is the lowest level of coupling) [6].

Static type checking, therefore, is a mixed blessing. It prevents many errors from entering a software system which might not otherwise be detected until run time. However, it limits the possible reuse of a module if a specific declaration environment must also be reused. Not only must the reused module be in the scope of those declarations, but so must its users. Further, those users are forced to communicate with that module using those external types rather than its own, making the resource master over its users instead of the other way around. It is common to use a set of global types to facilitate communication among the components of a program, however this practice prevents most, if not all, of the developed software from being used in any other program. The following is an abstraction of such a situation:

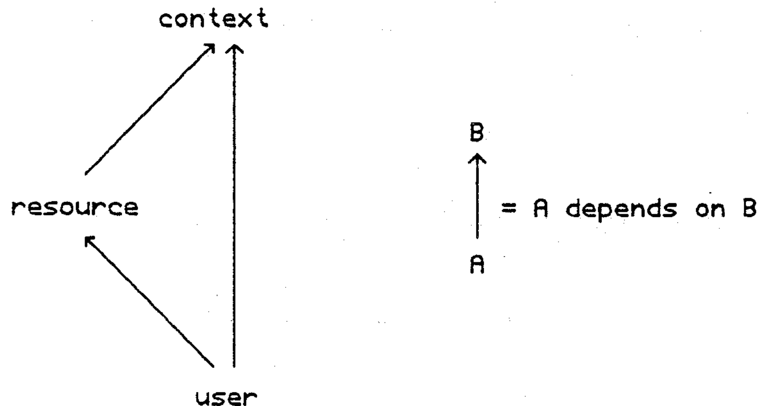
```
-- context:
package Global_Types is
  type Typ1 is ...
end Global_Types;

-- resource:
with Global_Types;
package Useful_Functions is
  procedure Proc (P1 : out Global_Types.Typ1);
end Useful_Functions;

-- user:
with Global_Types;
with Useful_Functions;
procedure User is
  Obj1 : Global_Types.Typ1;
begin
  Useful_Functions.Proc (Obj1);
end User;
```

The above illustrates the general case of a context-resource-user relationship. The dependencies among these three units can be illustrated with a directed graph. This form of dependency is related to the data binding dependencies discussed earlier. However, it is a dependence on type declarations and not on object declarations that is

important here. Nevertheless, the theory of data binding and cluster analysis could be extended to cover any relationship defined between modules, whether object dependence, type dependence, or any other. A graph of the dependency in the example above would be:



A resource does not always need full type information about the data it must access in order to accomplish its task. A common example would be a simple data base which stores and retrieves data but which does not take advantage of the information contained within that data. In cases such as this, it is possible to write or transform the resource so that any context or dependencies it requires are supplied by its users. Then, only the essential work of the module needs to remain. This "essence only" principle is the key to the transformations sought. Only the design of a module remains, with any details needed to produce the executing code, such as actual type declarations or specific operations on those types, being provided later by the end users of the resource. In languages such as Smalltalk which allow dynamic binding, this information is bound at run time. In Ada, where the compiler is obligated to perform all type checking and thereby eliminate many of the problems that can occur with dynamic binding, generics that are bound at compilation time can be used to free the text of a resource from depending directly on external type definitions.

Using the following package declaration and body, which is an abstraction of a hypothetical but structurally typical Ada package, one transformation is illustrated:

```
-- resource:
with Decls;
package Store is
```

```
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;

package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;
```

The above component can be transformed into the following one which has no dependencies on external declarations:

```
-- generalized resource:
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Typ) is
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

Note that, by naming the generic formal parameter appropriately, none of the identifiers in the code needed to change, and the expanded names were merely shortened to their simple names. This minimizes the handling required to perform the transformation (although automating the process would make this an unimportant issue). This transformation required the removal of the context clause, the addition of two lines (the generic part) and the shortening of the expanded names. The modification required to convert the procedure to a theoretically independent one constitutes a reusability measure. Formal rules for counting program changes have already been proposed and validated [7], and adaptations of these counting rules (such as using a lower handling value for shortening expanded names and a higher one for adding generic

formals) are being considered as part of this work.

Although the above illustration shows the context, the resource, and the user as library level units, declaration dependence can occur, and transformations can be applied, in situations where the three components are nested. For example, the resource and user can be co-resident in a declarative area, or the user can contain the resource or vice versa. For clarity, the examples in this paper will always show the components at the library level.

In addition to measuring the reusability of a unit by the amount of transformation required to optimize its independence, reusability can also be gauged by the amount of residual dependency on other units which cannot be eliminated, or which is unreasonably difficult to eliminate, by any of the proposed transformations. For any given unit, therefore, two values can be obtained. The first reveals the number of program changes which would be required to perform any applicable transformations. The second indicates the amount of dependence which would remain in the unit even after it was transformed. The original unit in the example above would score high on the first scale since the handling required for its conversion was negligible, implying that its reusability was already good (i.e., it was already independent or was easy to make independent of external declarations). After the transformation, there remain no latent dependencies, so the transformed generic would receive a perfect reusability score.

Note that the object of any reusability measurement, and therefore, of any transformations, need not be a single Ada unit. If a set of library units were intended to be reused together then the metrics as well as the transformations could be applied to the entire set. Whereas there might be substantial interdependence among the units within the set, it still might be possible to eliminate all dependencies on external declarations.

In the above example, one reason that the transformation was trivial was that the only operation performed on objects of the external type was assignment. Therefore, it was possible to replace direct visibility to the external type definition with a generic formal private type. A second example illustrates a slightly more difficult transformation

which includes more assumptions about the externally declared type. In the following example, indexing and component assignment are used by the resource.

Before transformation:

```
-- context
package A is
  type Item_Array is array (Integer range <>) of Natural;
end A;

-- resource
with A;
procedure B (Item : out A.Item_Array) is
begin
  for I in 1..10 loop
    Item (I) := 0;
  end loop;
end B;

-- user
with A, B;
procedure C is
  X : A.Item_Array (1..10);
begin
  B (X);
end C;
```

After transformation:

```
-- context
package A is
  type Item_Array is array (Integer range <>) of Natural;
end A;

-- generalized resource
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is array (Index range <>) of Component;
procedure Gen_B (Item : out Gen_Array);
procedure Gen_B (Item : out Gen_Array) is
begin
  for I in 1..10 loop
    Item (I) := 0;
  end loop;
end Gen_B;

-- user
with A, Gen_B;
procedure C is
  X : A.Item_Array (1..10);
  procedure B is new Gen_B (Natural, Integer, A.Item_Array);
begin
  B (X);
end C;
```

The above transformation removes compilation dependencies, and allows the generic procedure to describe its essential function without the visibility of external declarations. However, it also illustrates an important additional kind of dependence which can exist between a resource and its users, namely information dependence.

3.2. Information Dependence

In the previous example, the literal value 10 is a clue to the presence of information that is not general. In fact, even the appearance of the literal value 0 (the value assigned to the components) is not even necessarily general. Therefore, either of the following would be an improvement over the transformation shown above:

```
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is array (Index range <>) of Component;
procedure Gen_B (Item : out Gen_Array);
procedure Gen_B (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := Component'First;
  end loop;
end Gen_B;
```

- or -

```
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is array (Index range <>) of Component;
   Init_Val : Component := Component'First;
procedure Gen_B (Item : out Gen_Array);
procedure Gen_B (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := Init_Val;
  end loop;
end Gen_B;
```

Note that the last transformation allows the user to supply an initial value, but also provides the lowest value of the component type as a default. An additional refinement would be to make the component

type private which would mean that Init_Val could not have a default value. Information dependencies such as the one illustrated here are harder to detect than compilation dependencies. The appearance of literal values in a resource is often an indication of an information dependence.

3.3. Protocol Dependence

A third form of dependence has been identified, known as protocol dependence, where the user of a resource must obey certain rules to ensure that the resource behaves properly. For example, a stack which is used to buffer information between other users could be implemented in a not-so-abstract fashion by exposing the stack array and top pointer directly. In the following illustration, all users of the stack must follow the same protocol of decrementing the pointer before popping and incrementing after pushing, and not the other way around:

```
-- resource:
package Bad_Stack is
  Data : array (1..100) of Character;
  Pointer : Integer range 1..101 := 1;
end Bad_Stack;

-- Push user example:
with Bad_Stack;
use Bad_Stack;
procedure Push is
begin
  if Pointer <= 100 then
    Data (Pointer) := '@';
    Pointer := Pointer + 1;
  end if;
end Push;

-- Pop user example:
with Bad_Stack;
use Bad_Stack;
procedure Pop is
  X : Character;
begin
  if Pointer > 1 then
    Pointer := Pointer - 1;
    X := Data (Pointer);
  end if;
end Pop;
```

Notice that in addition to the protocol dependence, an information dependence is indicated by the appearance of the literal values 1, 100,

and 101 (but not the '@' since that is only data in the user code). Notice, also, that one could argue that the dependence on type Character from package Standard might be considered a declaration dependence which ought to be removed. Although the mention of Standard in a context clause is unnecessary, meaning the resource and its users are already compilation-independent, the identification and removal of the use of Character in the resource (by replacing it with a generic formal private type) would improve the generality of the resource.

3.4. Formalizing the Technique

The following is a formalization of the objectives of transformations which are needed to remove declaration dependence.

1. Let P represent a program unit.
2. Let D represent the set of n object declarations, d(1) . . d(n), directly referenced by P such that d(i) is of a type declared externally to P, but not in package Standard (for library units, this visibility must be obtained through a "with" clause).
3. Let O(1) . . O(n) be sets of operations where O(i) is the set of operations applied to d(i) inside P.
4. P is completely transformable if each O(i)(j) can be replaced with a predefined or generic formal operation.

The earlier example transformation is reviewed in the context of these definitions:

1. Let P represent a program unit.

   P = **procedure** B (Item : **out** A.Item_Array) **is** ...

2. Let D represent the set of n declarations, d(1) . . d(n), directly referenced by P such that d(i) is of a type declared externally to P, but not in package Standard.

   D = { A.Item_Array }

3) Let O(1) . . O(n) be sets of operations where O(i) is the set of operations applied to d(i) inside P.

   O(1) = { indexing by integer types, integer assignment to components }

4) P is transformable if each O(i)(j) can be replaced with a predefined or generic formal operation.

Indexing can be obtained through a generic formal array type. Although no constraining operation was used, the formal type could be either constrained or unconstrained since the only declared object is a formal subprogram parameter. Since component assignment is required, the component type must not be limited.

```
type Component is range <>;
type Index is range <>;
    followed by either:
type Gen_Array is array (Index) of Component;
    or:
type Gen_Array is array (Index range <>) of Component;
```

Notice that some operations can be replaced with generic formal operations more easily than others. For example, direct access of array structures (illustrated earlier) can generally be replaced by making the array type a generic formal type. However, direct access into record structures (using "dot" notation) complicates transformations since this operation must be replaced with a user-defined access function.

3.5. Experience with Transformations

To test the feasibility of the transformations proposed, a 6,000-line Ada program written by seven professional programmers was examined for reuse transformation possibilities. The program consisted of six library units, ranging in size from 20 to 2,400 lines. Of the 30 theoretically possible dependencies that could exist among these units, ten were required. Four transformations of the sort described above were made to three of the units. These required an additional 44 lines of code (less than a 1% increase) and reduced the number of dependencies from ten to five, which is the minimum possible with six units. Using one possible program change definition, each transformation required between two and six changes.

A fifth modification was made to detach a nested unit from its

parent. This required the addition of 15 lines and resulted in a total of seven units with the minimum six dependencies. Next, two other functions were made independent of the other units. Unlike the previous transformations which were targeted for later reuse, however, these transformations resulted in a net reduction in code since the resulting components were reused at multiple points within this program. Substantial information dependency was identified but remained among the units, however.

3.6. Reuse Guidelines Based on Dependencies

As with the data binding analysis, manual application of the principles and techniques of generic transformation and extraction has revealed several interesting and intuitively reasonable guidelines relative to the creation and reuse of Ada software.

- Avoid direct access into record components except in the same declarative region as the record type declaration.

Since there is no generic formal record type in Ada (without dynamic binding such a feature would be impractical) there is no straightforward way to replace record component access with a generic operation. Instead, user-supplied access functions are needed to access the components and the type must be passed as a private type. This is unlike array types for which there are two generic formal types (constrained and unconstrained). This supports the findings of others which assert that direct referencing of non-local record components adversely affects maintainability [8].

- Minimize non-local access to array components.

Although not as difficult in general as removing dependence on a record type, removing dependence on an array type can be cumbersome.

- Keep direct access to data structures local to their declarations.

This is a stronger conclusion than the previous two, and reinforces

the philosophy of using abstract data types in all situations where a data type is available outside its local declarative region. Encapsulated types are far easier to separate as resources than globally declared types.

- Avoid the use of literal values except as constant value assignments.

Information dependence is almost always associated with the use of a literal value in one unit of software that has some hidden relationship to a literal value in a different unit. If a unit is generalized and extracted for reuse but contains a literal value which indicates a dependence on some assumption about its original context, that unit can fail in unpredictable ways when reused. Conventional wisdom applies here, also, and it might be reasonable to relax the restriction to allow the use of 0 and 1. However, experience with a considerable amount of software which makes the erroneous assumption that the first index of any string is 1 has shown that even this can lead to problems.

- Avoid mingling resources with application specific contexts.

Although the purpose of the transformations is to separate resources from application specific software regardless of the program structure, certain styles of programming result in programs which can be transformed more easily and completely. By staying conscious of the ultimate goal of separating reusable function from application declarations, whether or not the functionality is initially programmed to be generic, programmers can simplify the eventual transformation of the code.

- Keep interfaces abstract.

Protocol dependencies arise from the exportation of implementation details that should not be present in the interface to a resource. Such an interface is vulnerable because it assumes a usage protocol which does not have to be followed by its users. The bad stack example illustrates what can happen when a resource interface requires the use of implementation details, however even resources with an appropriately abstract interface can export unwanted additional detail which can lead

to protocol dependence.

4. Conclusions and Future Directions

Two ongoing research projects seeking to characterize and measure aspects of reusability in Ada software have been described. The first project applies for the first time the results of earlier data binding work to the Ada language [4]. The second project motivates the transformation of Ada software to yield reusable components and describes a technique to accomplish that transformation. Both projects yield measures that provide visibility into Ada software for the purpose of identifying and evaluating portions of it for reuse.

By applying the principles described by each of the studies, several guidelines concerning the structure of reusable Ada code were revealed. All of the guidelines discovered so far are intuitively satisfying in that they complement rather than contradict conventional software engineering wisdom. Not all have been previously associated with promoting reusability, however. Instead, most have been previously recommended with respect to promoting readability and maintainability.

Several tools have been identified and described as a result of these studies. A tool to calculate data binding metrics could be used to highlight regions of reusability in existing software. Then, a tool to assist in the identification of inter-unit dependence as well as one to perform the necessary modifications to a unit in order to extract it as a reusable component could be used. Similar tools could also be used to characterize the software in order to provide feedback to the developers on the reusability of their code. Such tools would serve a purely measurement function. For example, the two methods of measuring reusability relating to the transformability of software (measuring both initial dependence and latent unremovable dependence) could be automated in order to guide a developer to write resources that are properly encapsulated and interfaces that are sufficiently abstract. Although this research does not propose to develop these tools, it is currently specifying their operation as a way of more clearly defining the techniques and principles involved in the study.

References

[1a] Basili, V. R. and Rombach, H. D. "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment", Technical Report, Dept. of Computer Science (CS-TR-2158) and Umiacs (UMIACS-TR-88-92), University of Maryland, College Park, MD 20742, December 1988.

[1b] Basili, V. R. and Rombach, H. D. "Towards a Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes", Technical Report, Dept. of Computer Science (CS-TR-2446) and Umiacs (UMIACS-TR-90-47), University of Maryland, College Park, MD 20742, April 1990.

[2] Basili, V. R. and Rombach, H. D. "The TAME Project: Towards Improvement-Oriented Software Environments", IEEE Transactions on Software Engineering, SE-14, June 1988.

[3] Doubleday, D. L., "ASAP: An Ada Static Source Code Analyzer Program," Dept. of Computer Science (CS-TR-1895), University of Maryland, College Park, MD 20742, August 1987.

[4] Hutchens, D. and Basili, V. R., "System Structure Analysis: Clustering with Data Bindings", IEEE Transactions on Software Engineering, SE-11, August 1985.

[5] Basili, V. R. and Turner A., "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, SE-1, December 1975.

[6] Myers G., "Composite/Strcutured Design", Van Nostrand Reinhold, New York, 1978.

[7] Dunsmore, H. E. and Gannon, J. D., "Experimental Investigation of Programming Complexity", Proc. ACM/NBS 16th Annual Technical Symposium on Systems and Software, Washington, D.C., June 1977.

[8] Gannon, J. D., Katz. E. E. and Basili, V. R., "Characterizing Ada Programs: Packages", Proc. Workshop on Software Performance, Los Alamos National Laboratory, Los Alamos, New Mexico, August 1983.