Measurement of Language Supported Reuse in Object Oriented and Object Based Software

(To appear in *The Journal of Systems and Software*)

James M. Bieman **Santhi Karunanithi**
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523

Abstract

A major benefit of object oriented software development is the support for reuse provided by object oriented and object based languages. Yet, measures and measurement tools that quantify such language supported reuse have been lacking. Comprehensive reuse measures, particularly for reuse with modifications, are necessary to evaluate the status of reuse in an organization and to monitor improvements. We develop a set of measurable reuse attributes appropriate to object oriented and object based systems and a suite of measures that quantify these attributes.

One of our major objectives is to measure reuse in software written in the object based language Ada. A set of suitable primitive reuse measures are expressed in Ada Reuse Tables. These tables support the flexible use of primitive measures in programs with nested packages and subprograms, and Ada generic packages. We designed and implemented a prototype Ada Reuse Measurement Analyzer (ARMA) to generate measurement values from Ada programs. ARMA produces a *reuse data representation* and a corresponding *forest representation* of an Ada system that contain the information necessary to produce the primitive measures. Developers can use the representations to produce customized reports to satisfy a wide range of measurement goals. We use ARMA to measure primitive reuse attributes for a set of industrial Ada software. We also show that ARMA can be used to generate a set of component access and package visibility measures.

1 Introduction

Developing software in a predictable time period with acceptable quality and reliability is a major problem in the computer industry. There is a high demand for software of increasing complexity and a need to modify and extend existing software systems. Finding solutions to this "software crisis" has proven to be very difficult [1].

An important step towards solving the software crisis is via software reuse. New software systems are often developed from scratch even though much of the code in a given system can be found in other similar systems [2, 3, 4]. Software reuse can potentially increase the quality and productivity of software development and maintenance. Software reuse limits the amount of

Address correspondence to J. Bieman, Department of Computer Science, Colorado State University, Fort Collins, CO 80523. (303)491-7096, Fax: (303) 491-6639, Email: bieman@cs.colostate.edu

Research partially supported by NASA Langley Research Center, the Colorado Advanced Software Institute (CASI) and CTA, Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

new software that needs to be produced and thus allows a greater focus on quality. The reuse of well tested software should result in greater reliability and less testing time. With reuse, software development becomes a capital investment.

A significant infrastructure is required for software reuse to make a real impact on the software process. This infrastructure can include programming languages that are designed to effectively support and promote software reuse. Object Oriented Programming Languages (OOPLs) and Object Based Programming Languages (OBPLs) are two classes of programming languages that have features to support reuse.

The two central abstraction mechanisms in object oriented software are data abstraction and inheritance. The control and communication mechanism is message passing rather than direct procedure or function invocation. Global variables are not directly accessed. Rather, variables are accessed via messages passed to objects.

An *object based* system is one that is built around data abstraction; an *object oriented* system also makes use of *inheritance*. Inheritance provides language support for a developer to modify a particular class to create a new class, the *subclass*, that behaves somewhat differently than the parent class, the *superclass*. The subclass can be modified by adding new state variables, adding new operations, called *methods*, and/or changing existing methods. When creating a subclass, a developer need only specify the differences from the superclass.

Inheritance is a powerful mechanism available in OOPLs to support reuse. Both OOPLs and OBPLs have additional features that can be used to promote software reuse. They support information hiding, the separation of a visible interface specifications from a hidden implementation body, and they support modular designs. Many OOPLs and OBPLs also support generic and/or polymorphic operations, operations that can be applied to arguments of varying types.

Though OOPLs and OBPLs are designed to effectively support and promote software reuse, users need to understand the software engineering principles to produce programs that are easy to reuse. Developers who lack a thorough familiarity with the principles of information hiding and data abstraction are unlikely to produce reusable programs. The measurement of reuse will help developers to characterize the reusability of programs, monitor current levels of reuse and help provide insight in developing software that is easily reused.

Proponents assert that a major benefit of object oriented or object based design and programming is the generation of reusable software components [5]. To support or refute such claims, we must be able to measure reuse in these systems. Current reuse measures are not directed toward the object oriented approach. We need new definitions of attributes, abstractions and measures related to data abstraction, information hiding and inheritance constructs.

Our objective is to identify a suite of measures for primitive reuse attributes appropriate to object oriented systems. The attributes of reuse measurement are internal product attributes related to properties of particular software documents [6]. Measure development follows the guidelines of scientific measurement principles as applied to software [7, 8, 9]. In order to derive reuse measures, we must identify and clearly specify the software reuse attributes, the documents to be measured, formal models of the reuse attributes, and a method for generating consistent numerical values from the documents and attributes.

Reuse can not be completely quantified by a single reuse measure. Primitive measures based on primitive reuse attributes are clearly components of the "level of reuse." Composite measures are often less sensitive than the primitive measures and must be defined carefully [6]. Thus, we avoid combining the primitive components into composites. The primitive components can later be used (or combined) to satisfy specific goals and questions of software developers.

In prior work, we defined classes of software reuse, identified important perspectives of reuse, proposed relevant reuse abstractions, and suggested reuse attributes and associated measures ap-

plicable to object oriented systems [10, 11]. In this paper, we extend this work by developing more detailed reuse measures. We focus extra attention on reuse with the OBPL Ada, especially in our reuse measurement tool development efforts (due to the interests of our research sponsors). We develop measurement tools, and test them on industrial software.

This paper is organized as follows. In Section 2, we describe reuse classifications and perspectives, product documents, and define a set of measurable reuse attributes appropriate to object oriented and object based systems. Section 3 describes the design of the *Ada Reuse Measurement Analyzer* (ARMA), developed for measuring reuse in Ada systems. In Section 4 we demonstrate ARMA by applying it to industrial software data, and we evaluate the measures and ARMA. Section 5 describes the relation between our work and software measurement principles, software reuse measures for procedural software, and other measures for object oriented systems. We summarize the results, give our conclusions, and propose directions for future work in Section 6. Appendix A contains the BNF that specifies the reuse data representation that is generated by ARMA.

2 Candidate Reuse Measures

Reuse can be measured along three axis: *public* or *private* [6], *verbatim* or *leveraged*, and *direct* or *indirect* [10]. *Public reuse* is reuse from a library of software developed externally, while *private reuse* is reuse of code within one project. *Verbatim reuse* is reuse without modifications. Reusing code from a library without change is verbatim reuse. *Leveraged reuse* is reuse with modifications. Leveraged reuse is accomplished when code from predefined classes is tailored to a new use. These modifications can be either *ad hoc* modifications (modifications not supported by the programming language) or more disciplined modifications (modifications made using language support.) Leveraged reuse is effectively measurable when it is supported by the language or programming environment. *Direct reuse* is reuse without going through an intermediate entity. *Indirect reuse* is reuse through an intermediate entity. When a module A uses another module B and a module C uses module A, then module C indirectly uses module B. The distance or level of indirection is the number of intermediate entities between a client (user) and a server (used). There may be different possible intermediate entities connecting a client and a server. So there may not be a unique distance of indirect reuse between two modules in a system.

In OOPLs, verbatim reuse is accomplished by object instantiation of predefined classes, by the use of predefined classes as class components, and calls to subroutines. Object oriented support of leveraged reuse via genericity and inheritance provides an enhanced ability to analyze and measure leveraged reuse. Generics are simply templates for classes or subprograms. They are general versions of processes that can be modified by parameters at compilation time. Though reuse through genericity is verbatim in the sense that all the procedures are the same, it is also a form of leveraged reuse as the template is modified according to generic parameter values. Hence, reuse through genericity is a separate reuse classification from reuse through inheritance (which is a form of leveraged reuse). Thus, we classify object oriented reuse as either *verbatim reuse*, *generic reuse*, or *leveraged reuse*.

Since visible interfaces are separated from hidden bodies in object oriented systems, the amount of direct reuse in the interface of a class will increase the indirect reuse of that class. Hence, the amount of reuse in interfaces and hidden bodies need to be measured separately.

2.1 Reuse Perspectives

Different reuse attributes are visible when reuse is examined from different perspectives. Consider a system where individual modules access some set of existing software entities. A program unit

being reused is considered a server and the unit accessing that program unit is considered a client. For example, when module M uses program unit S, M is a client and S is a server. Reuse can be observed from the perspectives of the server, the client, and the system. Each of these perspectives is relevant for the analysis and measurement of reuse in a system.

A set of potentially measurable attributes can be derived for OOPLs based on profiles of reuse from each perspective. In each perspective, reuse can be categorized as either verbatim, generic, or leveraged. As a result, we can define numerous measurable attributes. Because of the numerous candidate reuse measures that can be derived, they are presented here in a set of tables.

Client Perspective

The *client perspective* is the perspective of a new system or a new system component. The reuse analysis focuses on how a new class reuses existing library classes or other classes. A client can have verbatim reuse with a server class or global server method, can instantiate a generic server class, and/or inherit from a server class. Thus from a client perspective, measures of the number of servers reused, the number of times each server is reused, and the size of each server, etc. are important. Since an object or a method can be reused, the number of times a method in a server is invoked, and the size of that method are also relevant measures. The size of a reused component can be determined by source lines of code, number of bytes, function points and any relevant measure. Again, a client can reuse a server either directly or indirectly. Hence, the number of indirect servers, indirect clients, levels of indirection etc., can be measured. The concept of "level" of indirection can be applied to all verbatim, leveraged and generic reuse. There may be different possible paths connecting a client and a server. Thus there may not be a unique distance of the indirect reuse between a client and a server in a system. A count of number of paths between a particular client and a server gives a measure of indirect reuse.

Table 1, Table 2, and Table 3 give reuse measures from the client perspective. In each table, candidate measures are named, and, when necessary, a more specific definition is given.

Table 1 gives verbatim reuse measures. These measures include counts of the externally defined entities used by a client without modifications, and size measures of these entities. Such measures can indicate how much previously defined code was reused without modification by a client. We can determine the number of server classes referenced without modification and how useful the classes are by counting the number of object instances of the reused class that are created. The size measures indicate the reuse in terms of the length of code that is reused. Counts of the indirect server classes and instances provide quantitative information about servers that are reused by the immediate servers.

Table 2 describes a set of candidate generic reuse measures. The measures are similar to those of Table 1 except for the inclusion of counts and size indicators related to generic instantiation and use. In Table 2 the *# Server instance creations* refers to the number of different classes that are instantiated from generic templates. We also count the number of object instances from each class. The size of the generic parameter declaration is also included. With this added information, we can analyze the reuse of generic templates by a client.

Table 3 describes leveraged reuse measures from the client perspective. Leveraged reuse is supported in object oriented software via inheritance. Thus, the measures here focus on the structure of inheritance use. From the client perspective we are interested in quantifying the attributes of the client's ancestor classes. We count the ancestors (both immediate and distant ancestors), and the number of methods that are inherited. Thus, we can determine for a given client how much previously defined code was reused via inheritance.

Client perspective measures quantify the reuse of previously existing code by the client. They can be used to indicate how much reuse is taking place in new development. For further insight,

	Candidate Measures	Definition
1.	# Direct server classes	No. of classes included for verbatim usage.
2.	# Indirect server classes	No. of classes that direct servers have Using, Generic Instantiation, and Inheritance indirect relationships.
3.	# Server instance creations	No. of object instances of each server.
4.	# Server methods	No. of distinct server methods called by clients.
5.	# Server method instances	No. of calls to, or usages of each server method.
6.	# Global server methods	No. of non-generic global procedures/functions.
7.	# Global server instance	No. of usages of each global servers.
8.	Size of each server method	
9.	Size of server interface	
10.	Size of global definitions in server interface	
11.	Size of global definitions in server body	
12.	Size of each client method	
13.	Size of client interface	
14.	Size of global definitions in client interface	
15.	Size of global definitions in client body	
16.	# Paths to indirect servers	No. of paths connecting client and indirect servers.
17.	Length of paths to indirect servers	No. of edges in a path connecting client and indirect servers.

Table 1: Verbatim reuse measures from client perspective

these client perspective measures should be taken separately for the interface and body of the client.

Server Perspective

The *server perspective* is the perspective of the library or a particular library component. Given a class, the analysis focuses on how a server is being reused by all client classes in the system. A set of reuse measurements can be taken from the server perspective. These measurements can determine which library components are being reused and in what manner (verbatim, generic, leveraged, directly, indirectly). If the number of methods in a server class is larger, then the potential impact on children is larger, since children can inherit all the methods defined in the object. The number of children of a parent server class gives a measure of the potential influence that a class has on the design. Thus, from the server perspective, relevant measures include the number of clients, the number of times the server is reused by all clients, size of each client, size of each client method that invokes a server method, the count of server methods, the size of the parameter list of server methods, and the number of times each method in the server is invoked. Under the leveraged reuse classification, measures include the depth of inheritance tree, and the number of immediate sub-classes of a class in the class hierarchy. A server class which is generic in nature can be reused through class instantiation or inheritance and a server class which is non-generic in nature can be reused through object instantiation or inheritance. Since a server can be reused directly or through an intermediate entity, the number of indirect clients, and levels of indirection are also relevant measures.

Table 4 lists reuse measures from the server perspective under each reuse classification (verbatim, generic, and leveraged). One difference between the entries in Table 4 and those in Tables 1, 2, and 3 is that the words "server" and "client" are swapped. Another difference is that, from the server perspective in Table 4, we include measures of the use of individual server methods. The focus here is on how the server classes and methods are actually accessed by clients. These measures can be used to determine which components are reused more, and thus can help us evaluate the differences between heavily reused servers and the servers that are seldom reused.

	Candidate Measures	Definition
1.	# Direct server classes	No. of generic classes included for generic instantiation.
2.	# Indirect server classes	No. of classes that direct servers have Using, Generic Instantiation, and Inheritance indirect relationships.
3.	# Server instance creations	No. of generic class instances of each generic server.
4.	# Object instance creations	No. of object instances of generic class instances.
5.	# Server methods	No. of distinct server methods called by clients.
6.	# Server method instances	No. of calls to, or usages of each server method.
7.	# Global server methods	No. of global procedures/functions.
8.	# Global server instances	No. of usages of each global servers.
9.	Size of each server method	
10.	Size of server interface	
11.	Size of global definitions in server interface	
12.	Size of global definitions in server body	
13.	Size of generic declarations in server	Size of generic parameters declaration.
14.	Size of each client method	
15.	Size of client interface	
16.	Size of global definitions in client interface	
17.	Size of global definitions in client body	
18.	# Paths to indirect servers	No. of paths connecting client and indirect servers.
19.	Length of paths to indirect servers	No. of relations or edges in a path connecting client and indirect servers.

Table 2: Generic reuse measures from client perspective

	Candidate Measures	Definition
1.	# Direct server classes	No. of direct superclasses.
2.	# Indirect server classes	No. of classes that direct servers have Using, Generic Instantiation and Inheritance indirect relationships.
3.	# Indirect parent servers	No. of indirect super classes for client.
4.	# Direct server methods inherited	No. of methods from server class available for client.
5.	# Direct server methods extended	No. of methods in client extended from corresponding server methods.
6.	# Direct server methods overridded	No. of methods in client overridding corresponding server methods.
7.	# Direct server overloaded methods	No. of methods in client overloading server methods.
8.	Size of each direct server method that is reused / extended	
9.	Size of direct server interface	
10.	Size of global definitions in server interface	
11.	Size of global definitions in server body	
12.	Size of each client method	
13.	Size of client interface	
14.	Size of global definitions in client interface	
15.	Size of global definitions in client body	
16.	Paths to indirect servers	No. of paths connecting client and indirect servers.
17.	Length of paths to indirect servers	No. of edges in a path connecting client and indirect servers.
18.	Paths to indirect parent servers	No. of paths connecting client and indirect parent servers.
19.	Length of paths to indirect parent servers	No. of edges in a path connecting client and indirect parent servers.

Table 3: Leveraged reuse measures from client perspective

	Candidate Measures	Definition
	Verbatim	**Using Relationship measure**
1.	# Direct clients	No. of classes having verbatim usage of server.
2.	# Indirect clients	No. of classes that have Using, Generic Instantiation, and Inheritance indirect relationships with direct clients.
3.	# Direct client invocations of server	No. of object instances in all clients.
4.	# Client invocations of server method	No. of calls to each server method in all clients.
5.	Count of server methods	No. of methods declared in server.
6.	Size of parameter list of each server method	
7.	Size of server interface	
8.	Size of each method in server	
9.	Size of global definitions in server interface	
10.	Size of global definitions in server body	
11.	# Paths to indirect clients	No. of paths connecting server and indirect clients.
12.	Lengths of paths to indirect clients	No. of edges in a path connecting server and indirect clients.
	Generic	**Generic Instantiation measure**
1.	# Direct clients	No. of client classes instantiating the generic server.
2.	# Indirect clients	No. of classes that have Using, Generic Instantiation, and Inheritance indirect relationships with direct clients.
3.	# Server instantiations	No. of class instances in all direct clients.
4.	# Direct client invocations of server	No. of object instances in all direct clients.
5.	# Client invocations of server method	No. of calls to each server method in all clients.
6.	Count of server methods	No. of methods declared in server.
7.	Size of parameter list of each server method	
8.	Size of server interface	
9.	Size of each method in server	
10.	Size of global definitions in server interface	
11.	Size of global definitions in server body	
12.	Size of generic declarations in server	Size of generic parameters declarations.
13.	# Paths to indirect clients	No. of paths connecting server and indirect clients.
14.	Lengths of paths to indirect clients	No. of edges in a path connecting server and indirect clients.
	Leveraged	**Inheritance Relationship measure**
1.	# Direct clients	No. of direct subclasses.
2.	# Indirect clients	No. of classes that have Using, Generic Instantiation, and Inheritance indirect relationships with direct clients.
3.	# Indirect child clients	No. of clients that have inherited indirectly from server.
4.	# Client invocations of server method	No. of times a method in server is reused in all its clients.
5.	Count of server methods	No. of methods declared in server.
6.	Size of parameter list of each server method	
7.	Size of server methods	
8.	Size of server interface	
9.	Size of global definitions in server interface	
10.	Size of global definitions in server body	
11.	Paths to indirect clients	No. of paths connecting server and clients.
12.	Length of paths to indirect clients	No. of edges in a path connecting client and server.
13.	Paths to indirect child clients	No. of paths connecting server and child clients.
14.	Length of paths to indirect child clients	No. of edges in a path connecting a server and child client.

Table 4: Reuse measures from server perspective

System Perspective

The system perspective is a view of the reuse in the overall system, both servers and clients, and includes system-wide private reuse and system-wide public reuse. The system perspective characterizes overall reuse of library classes in the new system. Measurable system reuse attributes include:

- Percentage of the new system source text imported from the library. This data, which can be compiled from client perspective data, can be used to evaluate the effectiveness of an overall reuse program. Increases in the percentage of imported code can result in a more reliable system if the library software is well tested.
- Percentage of new system classes used verbatim from the library (client view) and percentage of library classes that are imported verbatim (server view). This data can be used to evaluate how directly useful a library is. Reuse is simpler if no changes to imported classes are needed.
- Percentage of new system classes derived from library templates and percentage of library templates that are imported. This information can be used to evaluate the usefulness of any polymorphism that is designed into library templates.
- Percentage of new system classes derived from library classes through inheritance and percentage of library classes that are inherited. The adaptability of the library classes can be evaluated. We can see if there is a good fit between the more general classes in the library and the specialized classes in the new system.
- The average number of verbatim, generic and leveraged direct and indirect clients for servers, and servers for clients. We can examine the distribution of reuse via each reuse mechanism throughout the system.
- The average length and number of paths between servers and clients for verbatim, generic and leveraged reuse. Indirect reuse is actually a form of coupling — it creates dependencies between system units. We recommend using measurement to monitor the pervasiveness of indirect reuse.

System perspective reuse measures provide an overall picture of the reuse in a system, and can be used to evaluate the effectiveness of a reuse program.

2.2 Product Documents

Reuse attributes are internal product attributes related to properties of particular software documents. These documents may be design documents or code documents. Design documents can be in the form of graph models such as call multigraphs, inheritance hierarchy graphs, and Booch diagrams [12]. Figure 1 is an example call multigraph, where each node represents a module and each edge represents an invocation. In Figure 1, module $M0$ calls modules $M1, M2$, and $M3$, module $M1$ calls module $M4$, module $M2$ calls modules $M3$ and $M4$, module $M3$ calls module $M5$ from two different locations.

Booch diagrams include class diagrams, object diagrams, module diagrams and process diagrams. Class diagrams and object diagrams describe the existence and meaning of key abstractions that form the design. A *class diagram* shows the existence of classes and their relationships in the logical design of a system. A single class diagram represents all or part of the class structure of a system. The three important elements of a class structure are classes, class relationships, and class

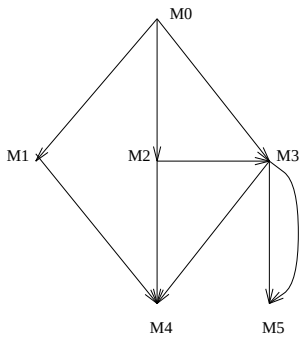

Figure 1: A call multigraph abstraction

utilities (global modules). The class relationships include inheritance, using, and instantiation relationships. Using relationships indicate the use of predefined classes as class components and object instantiations of predefined classes. Instantiation relationships indicate the class instantiations of predefined generic classes. Inheritance relationships indicate the connections between sub-classes and super-classes. An object diagram is used to show the existence of objects and their relationships in the logical design of a system. The purpose of each object diagram is to illustrate the semantics of key mechanisms in the logical design. These graph models are abstractions that can capture the various reuse measures. These graph models can also be derived from code documents.

2.3 Reuse Tables for Object Based Software

Reuse can not be completely assessed by a single measure. More specific attributes that contribute to the general notion of reuse need to be included in a comprehensive reuse profile. The proposed reuse measures are primitive and are flexible enough to be used in various analyses. To support flexible use of the primitive measures in our analyses of object based software, we specify the construction of a number of tables of primitive measures derivable from the graph models of reuse. Though an incidence matrix will serve the purpose of measurement, these tables are designed so that individual measures can be derived easily. These tables allow users to tailor their analysis to their own reuse goals. After creation of these tables, a user can use a spread sheet to analyze reuse.

Class Using Class Table (CUCT)

The CUCT supports the measurement of verbatim reuse in classes from both the client and server perspectives. When a class A has verbatim use of another class B, A can instantiate B and access the methods of B. Figure 2 is a CUCT table derived from the call multigraph in Figure 1. Each CUCT row represents a class in the measured system and each column represents a class that is used by a row class. Each table entry indicates both whether a row class 'uses' the associated column class, and the number of times a column class is instantiated in the row class. Thus, the $(1,2)$ in row $M3$ column $M5$ of Figure 2 indicates that $M3$ uses $M5$ and two instances of $M5$ are created by $M3$. By counting the number of entries in each row we can compute the number of direct server classes for each row client. The sum of the count of instance creations for each row is the total number of server instance creations. The sum of entries for each column is the number of direct clients of a server class. The sum of the count of instances for each column is the number of direct client invocations of a server. The number of uses of each method of the server class can be obtained from the MMRT table which will be described shortly. A CUCT table should be developed separately for interfaces of classes and hidden bodies of classes.

	M0	M1	M2	M3	M4	M5
M0	0	1 1	1 1	1 1	0	0
M1	0	0	0	0	1 1	0
M2	0	0	0	1 1	1 1	0
M3	0	0	0	0	1 1	1 2
M4	0	0	0	0	0	0
M5	0	0	0	0	0	0

Figure 2: CUCT table

Class Instantiating Generic Class Table (CIGCT)

The CIGCT supports the measurement of generic reuse from both the client and server perspectives. Like the CUCT, the CIGCT must be developed separately for interfaces and hidden bodies of classes. When a class A instantiates a generic class B, A reuses B. In the CIGCT, each row represents a class in the measured system and each column represents a generic class. Each cell has three fields and the first field indicates if a row class instantiates a column class. The second field indicates the number of times a generic column class is instantiated in a row class. The third field indicates the number of times a generic object is instantiated.

The row sum of first field gives the number of direct generic server classes, the row sum of second field gives the number of server instance creations and the row sum of third field gives the number of object instance creations. The column sum of first field gives the number of direct clients for each generic server, the column sum of second field gives the the number of server instantiations in direct clients and the column sum of third field gives the number of client invocations of server. The number of uses of each method of the column class can be obtained from the MMRT table.

Method Method Reuse Table (MMRT)

The CUCT and CIGCT indicate only class level verbatim and generic reuse. But a class method can reuse another class method and/or a global method. The MMRT indicates the method level reuse from the client and server perspectives. In the MMRT (except the first row and column), rows and columns are identified by method names. A method name can be a global procedure/function name or a procedure/function name in a class. Each cell in the first row indicates the size of the corresponding column methods and each cell in the first column indicates the size of the corresponding row methods. The methods of a class that are exported can be identified by the combination of the class name and method name. Each table entry indicates whether a row method calls a column method, and the number of times a column method is called in the row method.

The first row of the MMRT indicates the size of server methods and the first column indicates the size of the client methods. Thus, a user can measure the size of reused code in a new method. The count of the number of entries in each row is the number of server methods for each client method. The sum of the count of server instances for each row gives the total number of calls to server methods. The sum of entries for each column is the number of direct client methods. The sum of the count of server instances for each column is the number of direct client invocations of a server method.

Class Size Table (CST)

In addition to recording counts of various attributes, we also record the size of the software entities being examined. Thus, we need to measure the size of the reused and reusing code. In the MMRT, we record the size of methods. In addition, we record the size of the packages or classes using the CST. All rows in the CST are identified by class names. The table has four columns, one each for the size of class interface, the size of global definitions in the class interface, the size of global definitions in the class body, and the size of generic parameter declarations respectively. The last column is needed only for generic classes. Because visible interface specifications are separated from the hidden bodies in some languages (e.g., Ada and Modula-3), we separate these size measures.

Flexible Use of the Measurement Tables

The tables support the flexible use of the primitive measures allowing users to tailor analyses for their specific needs. For a given client class, a user can determine the amount of verbatim reuse from the CUCT table, the amount of generic reuse from the CIGCT table, and the amount of both verbatim and generic method reuse from the MMRT table. Users can also determine the size of the reused code from the MMRT and CST tables. Similarly, a user can measure the reuse of a library class from the tables.

A first step in automated measurement analysis is the collection of data item counts. The second step is the calculation of measurements through the formulation of the CUCT, CIGCT, MMRT and CST tables. The third step is the production of various reports based on obtained measurements. The form and content of these reports can be controlled by the user. The process is currently limited to the analysis of static reuse measurements.

2.4 Reuse Measurement in Ada

Ada is an OBPL that is designed to support large scale development with reusable components. It supports software portability and reuse. The current version of Ada does not support inheritance. Hence all the measures defined for object oriented systems except leveraged reuse measures are applicable for Ada. Ada supports software reuse through the following features:

- Program units: packages, subprograms, and tasks.
- Information hiding: private clauses that allow separation of visible interface specifications and hidden bodies.
- Strong type checking: compile time data type checking that improves the ability to reuse components without introducing context errors for data elements in the reused components.
- Generic program unit: parameterized templates that allow the generation of software components for specific types.
- Program libraries: separately compiled reusable program units.

A future version of Ada (Ada 9X) is likely to include additional features for generic parameters, overloading, and inheritance [13].

It is not possible to differentiate between the reuse of library classes from external systems, and internal reuse from within the new system. Ada does not provide any construct to differentiate between public library packages and local library packages. However in a particular programming environment the distinction between public reuse and private reuse can be made by those taking the measurements. In Ada, packages serve the purpose of classes in object oriented systems.

Reuse Table for Ada Systems

Ada allows nesting of packages and subprograms. The reuse tables given for object oriented systems have to be developed at each level of nesting. Also a package can have nesting of generic and non-generic packages. Hence, to support flexible use of primitive measures in Ada systems, we designed an *Ada Reuse Table* (ART) combining all the reuse tables in a different format. This table supports both verbatim and generic reuse measurements from both the client and server perspectives. Each row in the table represents a package or a subprogram in the measured system and each column represents a library unit that is used by a row package. Each row and column unit will have its size information along with its name. Each table entry has four parts. The first part gives the verbatim reuse count, the second part gives the generic instantiation count, the third part gives the generic object instantiation count and the fourth part gives the subprogram calls count. The sum of row entries of each part gives the total verbatim usage, total generic instantiation usage, total generic object instantiation usage and total subprogram calls respectively for that row client. Similarly, for a server, column sums give total verbatim and generic usage.

2.5 Using the Primitive Measures

A primary use of the primitive reuse measures is to quantitatively assess the status of software reuse in an organization. We assume that the organization manages a library of software components and encourages new software projects to maximize the reuse of library components. In this scenario, we see two major uses of the primitive reuse measures:

1. Organizations can use the primitive measures to evaluate the reuse of library components. Such an evaluation is conducted from the server perspective. The particular primitive measures that will be most valuable depend on what the organization wants to learn about the usage of the reuse library.

   The most general information will be of greatest value initially. The most general information for each library unit (class or package) includes counts of the number of direct clients in the verbatim, generic, and leveraged categories as shown in Table 4. These measures are counts of the number of clients that import the library unit without modification, the number of clients that instantiate each library unit, and the number of subclasses for each class in the library. These are the first measures in the verbatim, generic, and leveraged categories in Table 4.

   Simple counts of library unit access indicate the most useful components in the library. Further analysis (and more detailed measures) can help determine why some library units are frequently reused and others are seldom or never reused.

2. Organizations can evaluate the extent that software projects are able to reuse library (and other) components. Here, the focus is on how much reuse takes place in new development, and a client perspective is taken.

   Again, the most general information will be the most useful initially. Organizations are likely to be interested in the number of direct servers in the verbatim, generic, and leveraged categories shown as the first measures in Tables 1–3. The measures are counts of the number of imported classes or packages, the number of generic packages or templates that are instantiated, and the number of superclasses. More detailed information can be obtained if needed by using additional measures such as the number of server objects instantiated, the number of server methods referenced, or any of the other server perspective measures.

Initially, organizations are likely to find the most general measures to be the most useful. However, detailed analysis of a reuse program is likely to require some of the more focused measures included in the tables. Organizations are often interested in information concerning the quantity or size of reused components. Thus, measurements of reuse component size as described in Tables 1–4 will also be very useful.

One application of the primitive reuse measures is to analyze the visibility of program units. The package visibility measures of Gannon, Katz, and Basili can be derived from the server perspective measures [14]. The following visibility measures can be taken for each server:

- Used: Number of units where information from the server is accessed or changed.
- Current: Number of units where the server is currently visible.
- Available: Number of units where the server could be made visible (for example by using *with* clauses in an Ada system).
- Proposed: Number of units where the server is visible given the current unit structure and server is made visible in the most local position possible (for example using *with* clauses in their lowest possible nested positions in Ada units).

The visibility measures indicate whether software components are used whenever they are visible. Visible units that are not used can make it more difficult to maintain a system, and may indicate that these units are less reusable. The visibility measures may be increased by restructuring a program. Thus, these measures may be used to improve the design quality of a system.

We have initially applied the relevant primitive reuse measures to Ada software.

3 Ada Reuse Measurement Analyzer (ARMA)

We designed and implemented a prototype Ada Reuse Measurement Analyzer (ARMA) to demonstrate the utility of the candidate measures. Instead of developing Ada syntactic and semantic analyzers from scratch, we used the Anna tool set developed at Stanford [15] as a base for the ARMA tool design.

3.1 The Anna Tool Set

Anna is a language extension of Ada which includes facilities for formally specifying the intended behavior of Ada programs [16]. Anna is designed to augment Ada with precise machine-processable annotations so that formal methods of specification and documentation can be applied to Ada programs. The Program and Analysis Group at Stanford University has developed a prototype environment supporting the Anna language [15]. The tools implement a large subset of the Ada and Anna languages. The Anna tools are completely implemented in Ada and use the Descriptive Intermediate Attributed Notation for Ada (*DIANA*) to form the intermediate representation of Ada programs [17]. DIANA is designed to be especially suitable for communication between the front and back ends of a compiler. DIANA encodes the results of lexical, syntactic, and static semantic analysis in the form of an Abstract Syntax Tree (AST). The Anna tool set includes the following:

- Intermediate Representation Toolkit: Extended DIANA Formal Interface and Implementation packages (AST), Ada/Anna Parser, Ada/Anna Pretty Printer, AST Disk to Memory package, and AST Memory to Disk package.
- Ada/Anna Static-Semantic Rules Checker

- Annotation Transformer
- Portable Ada/Anna Testing and Debugging System
- Ada/Anna AST Browser

The Intermediate Representation toolkit defines the common internal representation used by all the Anna tools. The Parser parses Anna and hence Ada text files into an AST representation. The Pretty Printer generates an ASCII text given an AST. The AST Disk to Memory and AST Memory to Disk package perform AST disk-to-memory conversions, and memory-to-disk conversions. The Ada/Anna Static-Semantic Rules Checker inputs an AST and checks for the correctness of the static-semantics of the Ada and Anna code in the AST and updates the AST with semantic information. It works in both batch and incremental modes. The Annotation Transformer maps an Anna program to an equivalent Ada program. The AST Browser is an X-Windows based tool for graphically traversing and examining an Anna AST.

3.2 ARMA Design

The current version of ARMA does not use all of the available reuse information in providing reuse reports. However, ARMA is designed to be extended to conduct further analysis. ARMA consists of two parts - the Metric Generator and the Metric Analyzer. The Metric Generator extends the Anna tool set. Figure 3 shows the overall process used by ARMA. There are three phases to measuring reuse using ARMA.

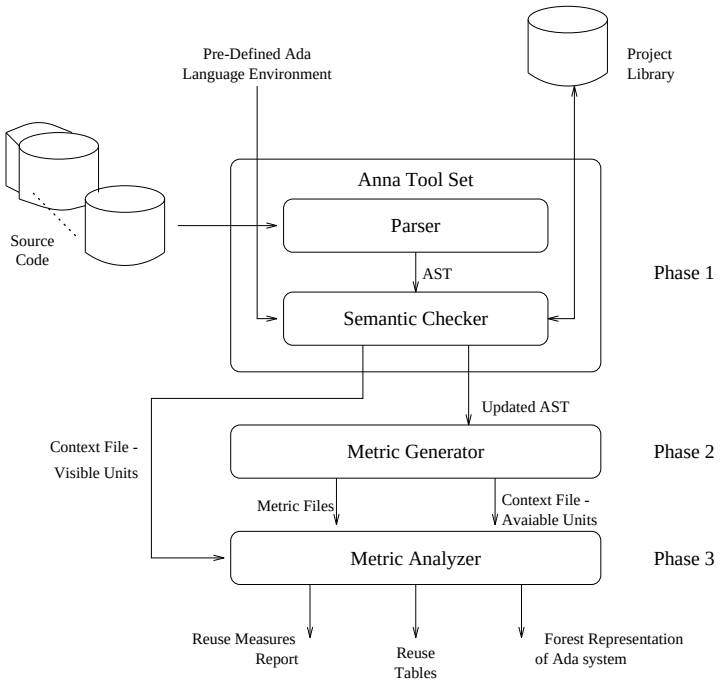

Figure 3: Ada Reuse Measurement Analyzer (ARMA) Design

Phase 1

In the first phase, the Anna tool set is used to create an AST with semantic information. The parser reads the Ada source code and produces an AST. The semantic checker inputs this AST and

information on the pre-defined language environment, and then updates the AST with semantic information. Phase I is the only phase that uses the source code. The Ada make file is used to identify the compilation order of the units of an Ada application. The semantic checker keeps a project library to use in processing later compilation units. Two of the outputs of the semantic checker, the AST and the library context file created for each compilable unit, are used in the later processing. The library context file identifies all the library units that are visible to a particular compilable unit.

Phase 2

The Metric Generator of ARMA inputs the updated AST and produces reuse measurements in a reuse data representation language. The Metric Generator consists of a *case* statement that recursively traverses the DIANA tree. The reuse data representation language allows the representation of all of the information necessary to generate reuse measurements, and identifies flow of control and flow of information throughout the system. For each compilable unit, ARMA creates a metric file in this language, and a library context file identifying all the library units that are made visible by a *with* clause. The BNF format of the reuse data representation language is given in Appendix A.

Phase 3

In the third phase, the Metric Analyzer of ARMA reads all of the metric files and library context files of the compilable units using the compilation order specified in the Ada make file. ARMA forms a forest of trees in which the root of each tree corresponds to packages or subprograms in an Ada application. Nested packages and subprograms form the sub trees of each top tree node. Each root node can have two children corresponding to the interface and hidden body of that node. If there are nested packages and subprograms in the interface or body, they will form sub trees at one level below the root node or at the body node of the root node. As the nesting level increases, this tree nesting also increases. Corresponding to the sub trees in the interface, the body branch also will have sub trees. Each node will also maintain information about the library units that are visible, the library units that are made available by a *with* clause, server units information, and client units information.

Figure 4 shows an example Ada system and its corresponding forest representation. The forest contains three trees, one for each of the two top level packages, packages X and Y, and one tree for the referenced server package *Text_Io*. The function *X.Add* appears in the interface node and in a body node of the tree for package X since the interface and bodies of the packages are in separate subtrees. The nested package *Y.Z* appears as a subtree in both the interface and body component of the tree for package Y.

All possible reuse measurement information can be derived from the forest representation for an Ada application. ARMA produces reuse tables as described in Section 2.4. The current system creates the reuse tables only for units at the top level of nesting. Since Ada allows nesting of packages and subprograms in many levels, a reuse table can be created for each nesting level. The outputs of Phase 3 are the forest representation (in ASCII form), a reuse table and measurement report. Currently, a reuse measurement report is produced which contains reuse measurements from the client and server perspective for each unit at top level in the Ada system.

3.3 Limitations of the Anna Tool Set

The Ada/Anna Static-Semantic Rules Checker has some severe limitations. The checker does not fully check the correctness of all Ada constructs, and hence the AST input to the Metric

(a)

```
package X is
   subtype My_Int is Integer range 1..100;
   function Add (f : in My_Int; l : in Integer) return Integer;
end X;

package body X is
   function Add(f : in My_Int; l : Integer) return Integer is
   begin
     return (f+l);
   end Add;
end X;


With X;
package  Y is
   package Z is
      size : X.My_Int;
      function Compute return Integer;
   end Z;
   procedure Print;
end Y;


With Text_Io;
package body Y is
   package body Z is
      function Compute return Integer is
      begin
        return X.Add (1,2);
      end Compute;
   end Z;
   procedure Print is
   begin
     Text_Io.Put_Line ("An Example Ada System");
   end Print;
end Y;
```

(b)

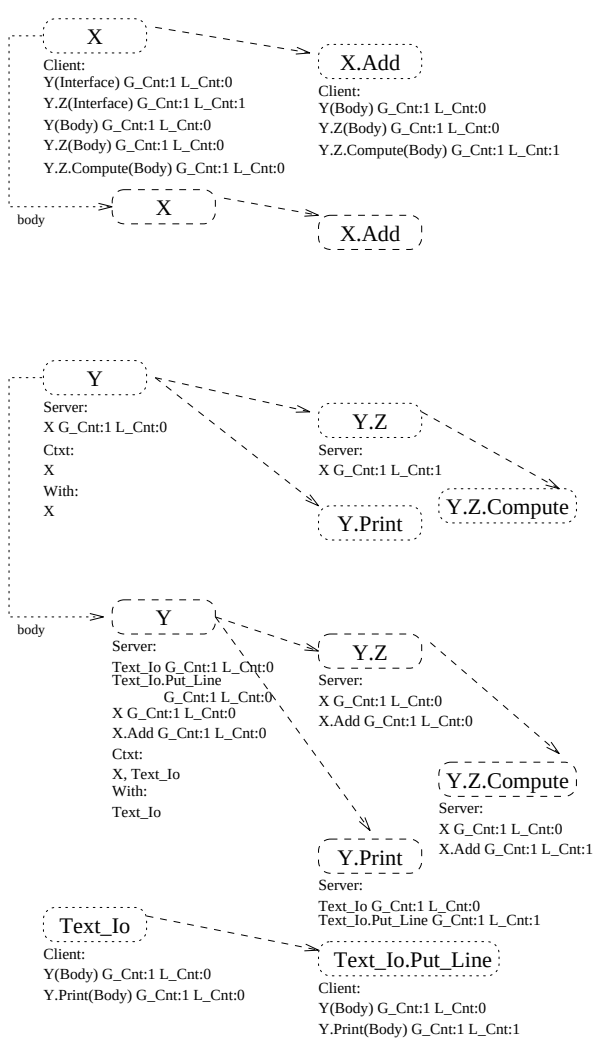

Figure 4: An Ada System (a) and its forest representation (b)

Generator of ARMA does not have full semantic information. The checker fails to process deeply nested package or subprogram declarations. For our analysis, the major problem with the Anna tool is that generic unit instantiations and renaming declarations are not analyzed completely by the checker. The second part of ARMA resolves the problems in generic unit instantiations and renaming declarations by forming the internal representation of the Ada system.

4 Using ARMA

We demonstrate ARMA by applying it to software provided by CTA, Inc. The data provides an opportunity to measure reuse from both the client and server perspectives.

4.1 The Test Data

The overall configuration of the software data is shown in Figure 5. The software data consists of 3 packages - *Global_Types*, *List_Manager*, and *Safe_Streams*, shown as tree roots in Figure 5. The Global_Types package contains the global data types declarations used in the data set. The

List_Manager package contains a generic package *List_Type* providing the *list* ADT. The List_Type uses type declarations given in Global_Types. The Safe_Streams package is a collection of different stream data type packages and subprograms. It contains 10 subpackages and 2 subprograms. These subpackages and subprograms use the type information provided by the package Global_Types and the generic package List_Type. The size of the data set is 5715 source lines containing comments

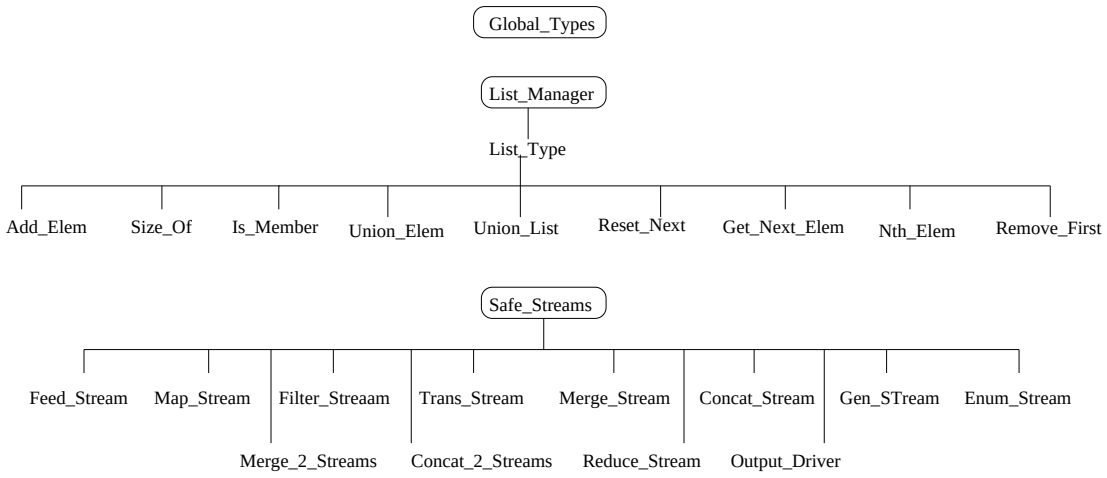

Figure 5: Data Set - 1

and 2935 lines without blank lines and comments (measured by not counting comments in the beginning of a line). These measures are count of lines in the Ada program files. ARMA measures LOC by counting number of semicolons. The size of the data set measured by ARMA is 2199 LOC.

4.2 ARMA Output

The complete output of the ARMA tests are quite lengthy, and are contained in a technical report [18]. The report includes the entire forest representation of the data set that was produced by ARMA.

Figure 6 gives a portion of the forest representation (in ASCII form) corresponding to the package interface *Global_Types* and the package body *Safe_Streams*. The term *Glob_Size* refers to the size of a library unit's import declaration. All clients of a unit are listed under `Client`; all the servers of a unit are listed under `Server`; all the library units that are made visible to a unit are listed under `Ctxt`; and all library units that are imported using *with* clause are listed under `With`. There are two parts to the usage measurement. The first part is the value of both direct and indirect usage, and the second part gives only the direct usage value. Each part contains the following set of four primitive measurements:

- `V:` Verbatim usage count,
- `G:` Generic instantiation count,
- `O:` Generic object instantiation count, and
- `C:` Number of calls made to the subprograms of the server.

The two parts are represented in the ASCII version of the forest representation in the form `V G O C :: V G O C`. For example, in Figure 6, *Safe_Streams.Merge_Stream (package body)* has verbatim use of *Global_Types* directly once and indirectly six times. Of the six indirect usage counts four of

```
* GLOBAL_TYPES (PAC_SPEC)                                Size: 5
  Client:
  LIST_MANAGER (PAC_SPEC)                                V  2 G 0 0 0 C 0 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE (GENERIC_PAC)                   V  2 G 0 0 0 C 0 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.SIZE_OF (FUNC_SPEC)             V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.NTH_ELEM (FUNC_SPEC)            V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  LIST_MANAGER (PAC_BODY)                                V  1 G 0 0 0 C 0 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE (PAC_BODY)                      V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS (PAC_SPEC)                                V  4 G 0 0 0 C 0 :: V 0 G 0 0 0 C 0
  SAFE_STREAMS.FEED_STREAM (GENERIC_PAC)                 V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS (PAC_BODY)                                V 15 G 0 0 0 C 0 :: V 0 G 0 0 0 C 0
  SAFE_STREAMS.FEED_STREAM (PAC_BODY)                    V  2 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.MERGE_STREAM (PAC_BODY)                   V  7 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.GEN_STREAM (GENERIC_PAC)                  V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.GEN_STREAM (PAC_BODY)                     V  2 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.MERGE_2_STREAMS (PAC_BODY)                V  2 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.REDUCE_STREAM (GENERIC_PROC)              V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.OUTPUT_DRIVER (GENERIC_PROC)              V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.OUTPUT_DRIVER (PROC_BODY)                 V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.REDUCE_STREAM (PROC_BODY)                 V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.MERGE_2_STREAMS.X_ADVANCE (PROC_BODY)     V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.GEN_STREAM.CHECK_BUFFER (PROC_BODY)       V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0
  SAFE_STREAMS.MERGE_STREAM.BUMP (FUNC_BODY)             V  4 G 0 0 0 C 0 :: V 4 G 0 0 0 C 0
  SAFE_STREAMS.MERGE_STREAM.X_ADVANCE (PROC_BODY)        V  2 G 0 0 0 C 0 :: V 2 G 0 0 0 C 0
  SAFE_STREAMS.FEED_STREAM.CHECK_BUFFER (PROC_BODY)      V  1 G 0 0 0 C 0 :: V 1 G 0 0 0 C 0

* SAFE_STREAMS (PAC_BODY)                                Size: 149 Glob_Size: 3
  Server:
  GLOBAL_TYPES (PAC_SPEC)                                V 15 G  0 0  0 C   0 :: V 0 G 0 0 0 C 0
  LIST_MANAGER (PAC_SPEC)                                V  0 G 14 0 24 C 128 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE (GENERIC_PAC)                   V  0 G 14 0 24 C 128 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.SIZE_OF (FUNC_SPEC)             V  0 G  0 0  0 C  37 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.REMOVE_FIRST (PROC_SPEC)        V  0 G  0 0  0 C  25 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.ADD_ELEM (FUNC_SPEC)            V  0 G  0 0  0 C  25 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.RESET_NEXT (PROC_SPEC)          V  0 G  0 0  0 C  13 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.GET_NEXT_ELEM (PROC_SPEC)       V  0 G  0 0  0 C  13 :: V 0 G 0 0 0 C 0
  LIST_MANAGER.LIST_TYPE.NTH_ELEM (FUNC_SPEC)            V  0 G  0 0  0 C  15 :: V 0 G 0 0 0 C 0
  DIRECT_IO (GENERIC_PAC)                                V  0 G  1 0  2 C   3 :: V 0 G 0 0 0 C 0
  DIRECT_IO.READ (PROC_SPEC)                             V  0 G  0 0  0 C   1 :: V 0 G 0 0 0 C 0
  DIRECT_IO.CLOSE (PROC_SPEC)                            V  0 G  0 0  0 C   1 :: V 0 G 0 0 0 C 0
  DIRECT_IO.DELETE (PROC_SPEC)                           V  0 G  0 0  0 C   1 :: V 0 G 0 0 0 C 0
  Ctxt:
  GLOBAL_TYPES
  DIRECT_IO
  LIST_MANAGER
  With:
  LIST_MANAGER
  DIRECT_IO
```

Figure 6: A Portion of the Forest Representation

them are from the subprogram *Safe_Streams.Merge_Stream.Bump* and two are from the subprogram *Safe_Streams.Merge_Stream.X_Advance*.

Table 5 gives the summary of reuse measurements at the top level (non-nested packages) using an *Ada Reuse Table* (described in Section 2.4). The table includes entries for the size of the packages and the number and kind of reuse interactions between clients and servers using the `V, G, O, C` measurements.

Ada allows nesting of packages and subprograms. Thus, the reuse measurements can be taken at each level of nesting. Table 6 gives the reuse measurements for the nested packages and subprograms at the first level. In these tables, the reuse measurements for Ada standard packages are not included though they are given in the forest representation. The measurements give an indication of the amount of reuse within each unit and also the amount of usage of each library unit. The tables can be created for each level of nesting and reuse at any level can be analyzed separately. The results show that the generic package *List_Manager.List_Type* is used extensively in the package body of *Safe_Streams*. This is expected since *list* ADT is a general purpose ADT that is used in many applications. The zero entries for Safe_Streams package interface indicate that the List_Manager is used only in the hidden body of Safe_Streams, and hence its use is not exported through the interface packages or operations of Safe_Streams. A person using Safe_Streams does not have to know about List_Manager usage in Safe_Streams. From the forest representation we find that the List_Manager package is made visible to Safe_Streams (package body) by a "with" clause. Hence List_Manager is also visible to all the subunits of Safe_Streams (package body) though it is not used by all of these subunits. From Table 6 we find that the subunits Safe_Streams.Enum_Stream (body), Safe_Streams.Reduce_Stream (body) and Safe_Streams.Output_Driver (body) do not use List_Manager even though List_Manager is visible to them.

The data generated by ARMA demonstrates a difference between the visibility of List_Manager and its use. Further analysis can be conducted by generating the visibility measures of Gannon, Katz, and Basili [14] for the List_Manager package:

- Used: 9 units
- Current: 13 units
- Available: 14 units
- Proposed: 9 units

UA(List_Manager) or perceived generality of List_Manager is calculated as $Used/Available = 0.64$ and the PC(List_Manager) or excess visibility of List_Manager is calculated as $Proposed/Current = 0.69$. Since UA(List_Manager) = .64, List_Manager is usually used when it is available — it is reused in 64% of the packages that have access to it. The PC(List_Manager) indicates the quality of the system structure rather than the reusability of List_Manager.

Visibility is just one attribute related to reuse that ARMA can evaluate. ARMA is certainly flexible enough to analyze numerous other measurable attributes that are related to reuse and reusability. We plan to use ARMA to process additional data sets to further evaluate the proposed measures, and to derive new measures. The main objective of the ARMA design is to generate all possible reuse information. The results of the Metric Analyzer can be used as input to another *report generator* phase to prepare different reports. ARMA has been designed to be extended for additional analyses.

Client		Server								Total			
		Global_Types Size: 5				*List_Manager* Size: 23							
	Size	V	G	O	C	V	G	O	C	V	G	O	C
Global_Types(Interface)	5	0	0	0	0	0	0	0	0	0	0	0	0
Global_Types(Body)	2	0	0	0	0	0	0	0	0	0	0	0	0
List_Manager(Interface)	23	2	0	0	0	0	0	0	0	2	0	0	0
List_Manager(Body)	90	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams(Interface)	290	4	0	0	0	0	0	0	0	4	0	0	0
Safe_Streams(Body)	1789	15	0	0	0	0	14	24	128	15	14	24	128
Total		22	0	0	0	0	14	24	128	22	14	24	128

Table 5: Summary of Results

4.3 Further Limitations of the Anna Tool Set

While testing ARMA, we found several limitations to the Anna tool set, in addition to the limitations that we have already described. The Anna semantic checker cannot process nested packages correctly. Some of the Anna errors can be ignored while many others require some program manipulation. As a result, we had to modify some of the test data (without changing its reuse characteristics) for it to be processed by the Anna semantic checker. For example, some type declarations in the body of *Safe_Streams.Feed_Stream* were moved to the private part of the corresponding interface. Thus, until these problems with Anna are corrected, running ARMA will be time consuming since ARMA depends on the output of semantic checker. Note that the aim of this research is to develop a prototype tool to evaluate our ideas concerning reuse measurement. ARMA is not intended to be a production tool.

The problems that we encountered when we used part of the Anna tool set to build ARMA demonstrate the problems that many developers face when they reuse software.

4.4 ARMA and Reuse Activities

The information provided by ARMA can be used to identify reusable components and to monitor the extent that new development reuses existing code. ARMA can be used to generate Ada Reuse Tables as shown in Table 5. With this table, the most commonly reused server components can be quickly identified. In addition, such tables also show the reuse by clients. Tabulation of the four principal primitive measures are included (V, G, O, C). Additional information, on any level of detail or nesting, is contained in the forest representation generated by ARMA.

With the quantitative measurements provided by ARMA or a similar tool, an organization can evaluate the quality of a reuse library and a reuse process. For example, an organization may discover that certain packages are rarely used even though they appear to be very useful. These packages may not be reused because developers are not aware that they exist, they may be nested too deeply to be noticed, or the documentation may be inadequate. After improvements are made, the reuse can be monitored by ARMA to determine whether the reuse of these packages has increased. Thus, we can quantitatively determine if the redesign of the packages or changes in the process have been effective.

An organization can also use the reuse measures to monitor the success of a reuse program in promoting reuse in new development. The measures can quantify precisely how much and what kind of reuse is taking place. An organization can determine whether process changes have been effective in increasing reuse by clients.

Over time, the kind of detailed data generated by ARMA could lead to important insights

Client		**Server**								**Total**			
		Global_Types Size: 5				*List_Manager* Size: 23							
	Size	V	G	O	C	V	G	O	C	V	G	O	C
List_Manager. List_Type(Gen.Int)	19	2	0	0	0	0	0	0	0	2	0	0	0
List_Manager. List_Type(Body)	89	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams. Feed_Stream(Gen.Int)	23	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams. Map_Stream(Gen.Int)	25	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Filter_Stream(Gen.Int)	24	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Merge_Stream(Gen.Int)	23	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Concat_Stream(Gen.Int)	23	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Trans_Stream(Gen.Int)	27	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Gen_Stream(Gen.Int)	17	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams. Enum_Stream(Gen.Int)	26	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Merge_2_Streams(Gen.Int)	33	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Concat_2_Streams(Gen.Int)	33	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Reduce_Stream(Gen.Int)	16	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams. Output_Driver(Gen.Int)	16	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams. Feed_Stream(Body)	268	2	0	0	0	0	2	3	21	2	2	3	21
Safe_Streams. Map_Stream(Body)	120	0	0	0	0	0	1	2	6	0	1	2	6
Safe_Streams. Filter_Stream(Body)	124	0	0	0	0	0	1	2	6	0	1	2	6
Safe_Streams. Marge_Stream(Body)	181	0	0	0	0	0	3	4	26	0	3	4	26
Safe_Streams. Concat_Stream(Body)	168	0	0	0	0	0	2	3	21	0	2	3	21
Safe_Streams. Trans_Stream(Body)	146	0	0	0	0	0	1	2	10	0	1	2	10
Safe_Streams. Gen_Stream(Body)	177	0	0	0	0	0	2	4	20	0	2	4	20
Safe_Streams. Enum_Stream(Body)	94	0	0	0	0	0	0	0	0	0	0	0	0
Safe_Streams. Merge_2_Streams(Body)	157	0	0	0	0	0	1	2	6	0	1	2	6
Safe_Streams. Concat_2_Streams(Body)	172	0	0	0	0	0	1	2	12	0	1	2	12
Safe_Streams. Reduce_Stream(Body)	16	1	0	0	0	0	0	0	0	1	0	0	0
Safe_Streams. Output_Driver(Body)	17	1	0	0	0	0	0	0	0	1	0	0	0
Total		22	0	0	0	0	14	24	128	22	14	24	128

Table 6: Summary of Results with one level nesting

concerning software reuse and reusability. Long term data might indicate that certain kinds of design structures are more easily reused. For example, we might be able to determine the relative reusability of generic or templates structures, and we might learn how useful inheritance really is.

5 Related Work

Measurement techniques used in the physical sciences lay the foundation for practical software measurement [7, 8, 9, 6, 7]. A measure allows us to numerically characterize intuitive attributes of software objects and events. We need a clear definition and understanding of a software attribute before we can define measures. We must be able to determine that one software entity has more or less of some attribute before we can use measurement to assign a numerical quantity of the attribute to the entity. The orderings of the software entities implied by the measurement must also be consistent.

5.1 Reuse Measurement for Traditional Software

Many product reuse measures are based on only one attribute, program size or length. Conte [19], Boehm [20], Bailey [21], and Fenton [6] describe reuse measures that are based on comparisons between the length or size of reused code and the size of newly written code in particular software products. Modifications to reused code are not considered. Conte's reuse measure is used to estimate coding effort. Reuse reduces coding effort and, thus, reuse can affect effort estimation. Boehm and Bailey also use the size of reused code to adjust cost predictors.

Fenton proposed a measure of reuse based on the dependencies in an associated call graph. Fenton's reuse measures are *public reuse* and *private reuse*. Public reuse is defined as "the proportion of a product which was constructed externally." To use such a measure one must be able to clearly distinguish between the components that are from external sources and the components that are completely new. Private reuse is defined as the "extent to which modules within a product are reused within the same product." Using the call graph as an abstraction, Yin and Winchester measure the private reuse as $r(G) = e - n + 1$ where e is the number of edges and n is the number of nodes in the call graph [22]. Selby addresses the measurement of reuse with modifications [23]. He classifies modules into categories based on the percentage of new versus reused code in a module. The categories are completely new modules, reused modules with major revisions ($\geq 25\% changed$), reused modules with slight revisions (< 25 % changed), and modules that are reused without change. Thus a count of the number of modules in each category provides a measure of reuse.

The above reuse measures are designed for procedural rather than object oriented or object based software. The object oriented and object based approaches are fundamentally different from the traditional procedural/functional approach that separate data from procedures. Software measures developed with traditional methods in mind do not direct themselves to notions such as classes, inheritance, encapsulation, message passing, and generics.

5.2 Measurement of Object Oriented and Object Based Software

Chidamber and Kemerer [24] propose a set of class level measures specifically developed for measuring elements contributing to size and complexity of object oriented design. These measures include weighted methods per class, depth in inheritance tree, number of sub-classes subordinated to a class in the class hierarchy, and the cardinality of the set of methods available to the object. They also propose measures of some aspects related to the coupling between objects and the lack of cohesion in methods.

Li and Henry add additional measures to those developed by Chidamber and Kemerer [25]. They add a class level measure of coupling through message passing (defined as the number of send-statements defined in a class), and coupling through an abstract data type defined as the number of abstract data types defined in a class. They also define several class size measures including the number of local methods and the number of semicolons (clearly a language-dependent measure). They also show that the measures can be statistically related to maintenance effort.

Sheetz, Tegarden, and Monarchi derive a set of primitive counts of attributes of object oriented code at several levels [26, 27]. The measures that are unique to object oriented software are those related to the class level (called the object-level in [26, 27]) and the inheritance structure. Measures at the class level include the number of classes that use or are used by a class, unique messages received/sent, subclasses, and superclasses. Other class level measures include class-to-root depth, class-to-leaf depth, the number of properties defined in a class with the same name as an inherited property, and the number of properties defined in multiple parents of a class with the same name. The inheritance level measures include the maximum depth of the inheritance hierarchy, the maximum number of classes at any one level, and the number of inheritance links. The authors provide a model of "complexity" based on the measures, and some intuition concerning the relationship between the measures and coupling and cohesion; they use this model to combine the primitive measures into composites.

Lake and Cook developed a tool that counts various attributes in C++ programs [28]. The tool can generate the inheritance hierarchy tree and indicate the number of classes at each level, the depth of a class in the tree, and the number of sub-classes. They use the tool in an experiment to determine the relationship between class depth in the inheritance tree and the ability of programmers to perform maintenance tasks. The results indicate that it is easier for programmers to perform tasks on classes at or near the root of the inheritance hierarchy tree. These results are preliminary, since the study included only eleven subjects.

Gannon, Katz, and Basili [14] show that a good background in software engineering principles is necessary to use the full capabilities of the Ada language. They propose three measures for Ada packages: a component access measure and two package visibility measures UA and PC. A component access is a reference to a data type. The component access measure is defined as the ratio of component accesses of objects with non-locally defined data types to lines of text in a program unit. The UA measure represents the ratio of units in which a package is accessed to those in which it could be made available by adding a `with` clause, given the current unit structure. The PC measure is defined as the ratio of the number of units in which a package must be visible to those in which it is visible.

Gannon et al. claim that the component access measure measures the units resistance to changes. They also claim that the package visibility measure UA indicates the perceived generality of the package. PC directly indicates the success of design decision to minimize visibility. The component access measure may not be applicable for object oriented programs written with private data types. However, the measures UA and PC are applicable for programs written in OOPLs as well as OBPLs.

The research reported by Chidamber and Kemerer [24], Li and Henry [25], Sheetz, Tegarden, and Monarchi [26, 27], Lake and Cook [28], and Gannon, Katz, and Basili [14] represent just a sample of the ongoing research into object oriented and object based software measurement.

5.3 Our Reuse Measurement Approach

We focus our efforts directly on the problem of measuring reuse on systems with respect to the unique aspects of the object oriented and object based approaches. We do not seek only a few simple reuse measures or measures that combine primitive quantities into composite measures.

Our approach is to focus on the goal of determining how to measure the level or quantity of reuse, and we find no one simple or composite measure. We find many measurable reuse attributes that can be assessed in a variety of ways. We also develop measures of the reuse when the reused software is modified for new uses.

We feel that we will have the greatest success in defining useful software measures, when a clear goal for measurement is determined. This approach is consistent with *Goal/Question/Metric* paradigm of Basili and Rombach [29]. Our goal is to measure the quantity of reuse in object oriented and object based systems in order to answer relevant questions about reuse. Thus the array of reuse measures that we have defined can be used for maximum advantage in a variety of reuse programs.

6 Conclusions

In this paper, we identify different reuse attributes pertaining to object oriented and object based software systems, develop models or abstractions for capturing the reuse attributes, and derive a suite of measures to quantify the attributes.

We find that reuse is not simple to classify and measure. There are many ways to look at reuse and many ways that reuse can be measured. Reuse measurements are classified along three dimensions: *public* or *private*, *verbatim* or *generic* or *leveraged*, and *direct* or *indirect* reuse. Public reuse is reuse of externally constructed software and private reuse is reuse of software within a application product. Verbatim reuse is reuse without modifications. Generic reuse is reuse of generic templates and leveraged reuse is reuse with modifications supported through inheritance. Direct reuse is reuse without going through an intermediate entity and indirect reuse is reuse through an intermediate entity.

Reuse can be measured from the perspective of the client, the server, or from the system. From the client perspective, we can learn the extent that new development actually makes use of old code. From the server perspective, we can evaluate how a reuse library is used. From the system perspective, we can analyze the overall interactions between clients and servers. A number of potentially measurable attributes are derived based on these perspectives. Many of these attributes can be quantified with simple counts such as the number of direct or indirect clients or servers, the number of server class instances, the number of object instances, the number of calls to a method in the server, the number of library units that are visible to a unit, and the number of library units imported explicitly in a unit.

To demonstrate the proposed measures, we designed and implemented the prototype Ada Reuse Measurement Analyzer (ARMA). We use ARMA in an initial empirical evaluation of the reuse of a commercial software system supplied by CTA, Inc.

ARMA was built by extending the Anna-I tool set developed at Stanford [15], using a strategy aiming for maximum flexibility. We wanted a tool that can generate a suite of primitive measurements. These primitive measures can then be used to formulate more focused analyses that answer questions concerning specific measurement goals. ARMA produces a *reuse data representation* and a *forest of trees* representation that contain the information necessary to produce the primitive measures for an Ada system.

Developers can use the reuse data and forest representations to produce customized reports to satisfy specific goals. For example, ARMA can generate quantitative information about the visibility of packages and component access, using measurements defined by Gannon, Katz, and Basili [14]. A variety of other existing and yet to be defined measures can be calculated from the primitive measures and data representations. This flexibility increases the usefulness of the measurement tools, since processes and measurement needs are evolving.

Results from the tool testing provide initial profiles of software reuse. Our initial measurements also provide insight into what needs to be measured in future. Some of the insights are as follows:

- Having a single data type per package visible rather than more than one seems to increase the reusability of that package. This structure may also decrease the complexity of that Ada system. Distributing representation information of an object using more than one data type rather than centralizing it in a single data type will make programs more difficult to change. Changes in representation could involve changes in many compilation units.

- Too many levels of nesting of units may affect reuse since one has to understand the nesting levels of units to reuse some nested unit.

The measures obtained in this paper quantify only primitive properties of reuse. We do not combine them to form composite measures. Thus, one possible extension of this work will be to analyze primitive reuse measures to produce composite measures that can be closely tied to actual reuse and *reusability* in object oriented systems.

We are currently developing and testing tools for measuring reuse attributes in programs implemented in C++. We will use these tools to study how inheritance is actually being used for reuse in commercial software development.

Another problem that we are studying is the conflict between reuse and coupling. Many of the language supported reuse mechanisms will tend to increase coupling. Yet the current rule of thumb is that tightly coupled modules are more difficult to maintain, understand and modify. We hope to develop guidelines to help maximize reuse with minimal negative impact on coupling.

Though object oriented languages have been designed to support and promote reuse, programmers need to use proper design methods to produce programs with maximum reuse. Tools that measure attributes related to reuse in software systems can identify properties that make software more reusable. Such empirical results can help determine the amount of reuse in existing systems and identify the most frequently reused software components. Reuse measurement will also help users gain insights to help develop software that is easily reused.

Acknowledgements

This research is partially supported by the NASA Langley Research Center, the Colorado Advanced Software Institute (CASI) and CTA, Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

We thank the anonymous referees for their careful reviews. Their comments greatly improved both the content and the presentation.

References

[1] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[2] T. Biggerstaff and A. Perlis, editors. Special Issue on software reuseability. *IEEE Trans. Software Engineering*, volume SE-10(5). 1984.

[3] T. Biggerstaff and A. Perlis, editors. *Software Reusability Vols. I, II.* ACM Press. Addison-Wesley, 1989.

[4] T. Biggerstaff and C. Richer. Reusability framework, assessment, and directions. In T. Biggerstaff and A. Perlis, editors, *Software Reusability Vols. I, II.* ACM Press. Addison-Wesley, 1989.

[5] B. Meyer. Reusability: The case for object oriented design. *IEEE Software*, 4(2):50–64, Mar 1987.

[6] N. Fenton. *Software Metrics A Rigorous Approach.* Chapman & Hall, London, 1991.

[7] A. Baker, J. Bieman, N. Fenton, A. Melton, and R. Whitty. A philosophy for software measurement. *Journal of Systems and Software*, 12(3):277–281, July 1990.

[8] A. Melton, D. Gustafson, J. Bieman, and A. Baker. A mathematical perspective for software measures research. *Software Engineering Journal*, 5(5):246–254, 1990.

[9] N. Fenton and A. Melton. Deriving structurally based software measures. *Journal of Systems and Software*, 12(3):177–187, July 1990.

[10] J. Bieman. Deriving measures of software reuse in object-oriented systems. Proc. BCS Workshop on Formal Aspects of Measurement (1991). In T. Denvir, R. Herman, and R. Whitty, editors, *Formal Aspects of Measurement*, pages 63–83. Springer-Verlag, 1992.

[11] S. Karunanithi and J. Bieman. Candidate reuse metrics for object oriented and Ada software. *Proc. IEEE-CS Int. Software Metrics Symposium*, pages 120–128, May 1993.

[12] G. Booch. *Object oriented design with applications.* The Benjamin/Cummings Publishing, Inc., Menlo Park, CA, 1991.

[13] S. Cohen. Ada 9x project report: Ada support for software reuse. Technical Report SEI-90-SR-16, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, Oct 1990.

[14] J. Gannon, E. Katz, and V. Basili. Metrics for Ada packages : An initial study. *Communications of the ACM*, 29(7):616–623, 1986.

[15] G. Mendal. The Anna-I user's guide and installation manual, version 1.4 edition. Technical Report ERL 456, Stanford University, Computer Systems Lab, Stanford, CA, Sept 1992.

[16] David C. Luckham, editor. *Programming with Specifications: An Introduction to Anna, A language for specifying Ada programs.* Springer-Verlag, 1990.

[17] G. Goos, W. Wulf, A. Evans Jr., and K. Butler. *DIANA, An Intermediate Language for Ada*, volume 161. Springer-Verlag, 1983.

[18] S. Karunanithi and J.M. Bieman. Measuring software reuse in object oriented systems and Ada software. Technical Report CS–93–125, Computer Science Dept., Colorado State Univ., Fort Collins, CO, 1993.

[19] S. Conte, H. Dunsmore, and V. Shen. *Software Engineering Metrics and Models.* The Benjamin/Cummings Publishing, Inc., Menlo Park, CA, 1986.

[20] B. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[21] J. Bailey and V. Basili. A meta-model for software development resource expenditures. In *Proc. 5th Int. Conf. Software Engineering (ICSE-5)*, pages 107–116, 1981.

[22] B. Yin and J. Winchester. The establishment and use of measures to evaluate the quality of system designs. In *Proc. Software Quality and Assurance Workshop*, pages 45–52, 1978.

[23] R. Selby. Quantitative studies of software reuse. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Vol. II Applications and Experiences*, pages 213–233. Addison-Wesley, 1989.

[24] S. Chidambar and C. Kemerer. Towards a metrics suite for object oriented design. In *Proc. OOPSLA*, pages 197–211, 1991.

[25] W. Lei and S. Henry. Maintenance metrics for the object oriented paradigm. *Proc. IEEE-CS Int Software Metrics Symp.*, page to appear, May 1993.

[26] S. D. Sheetz, D. P. Tegarden, and D. E. Monarchi. Measuring object-oriented system complexity. *Proc. 1st Workshop on Information Technologies and Systems*, December 1991.

[27] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi. A software complexity model of object-oriented systems. *Decision Support Systems: The International Journal*, to appear.

[28] A. Lake and C. Cook. A software complexity metric for C++. Technical Report 92-60-03, Computer Science Dept., Oregon State Unviersity, Corvallis, Oregon, 1992.

[29] V. Basili and H. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Trans. Software Engineering*, SE-14(6):758–773, June 1988.

A BNF for the Reuse Data Representation Language

```
compilation_unit          ::= library_unit | secondary_unit
library_unit              ::= subprogram_declaration | package_declaration | generic_declaration |
                              generic_instantiation | subprogram_body
secondary_unit            ::= library_unit_body | subunit
library_unit_body         ::= subprogram_body | package_body
subunit                   ::= Subunit parent_unit_name < cr > proper_body
proper_body               ::= subprogram_body | package_body | task_body
subprogram_declaration    ::= proc_declaration | func_declaration
proc_declaration          ::= proc_specification
                              end_proc_specification
proc_specification        ::= proc_name < cr > [formal_part] [import_part]
end_proc_specification    ::= End_Of proc_name size_declaration
func_declaration          ::= func_specification
                              end_func_specification
func_specification        ::= func_name < cr > [formal_part]
                              result_formal_part [import_part]
end_func_specification    ::= End_Of func_name size_declaration
proc_name                 ::= PROC_SPEC unit_name
func_name                 ::= FUNC_SPEC unit_name
formal_part               ::= par_specification [par_specification]
par_specification         ::= mode par_name:type_name=boolean < cr >
mode                      ::= Mode_In | Mode_Out | Mode_In_Out
boolean                   ::= Y | N
result_formal_part        ::= Res type_name < cr >
size_declaration          ::= Size value < cr >
import_part               ::= import_par_specification [import_par_specification]
import_par_specification  ::= Par_Imp type_name < cr >
package_declaration       ::= package_specification
                              end_package_specification
package_specification     ::= pac_name < cr >
                              [basic_declaration]
                              [Priv < cr > [basic_declaration]
                               End_Of Priv size_declaration]
end_package_specification ::= End_Of pac_name size_declaration
pac_name                  ::= PAC_SPEC package_name
package_body              ::= pac_body_name < cr >
                              [basic_declaration]
                              body_detail
                              End_Of pac_body_name size_declaration
pac_body_name             ::= PAC_BODY package_name
generic_declaration       ::= Gen_Type generic_specification
generic_specification     ::= proc_specification
                              generic_par_list
                              End_Of Gen_Type proc_name size_declaration |
                              func_specification
                              generic_par_list
                              End_Of Gen_Type func_name size_declaration |
                              package_specification
                              generic_par_list
                              End_Of Gen_Type pac_name size_declaration
generic_par_list          ::= Generic_par < cr >
                              gen_par_list [gen_par_list]
                              End_Of Generic_Par size_declaration
                              [generic_par_import]
gen_par_list              ::= gen_mode | type_use | subprogram_declaration
gen_mode                  ::  Gen_Mode_In type_name < cr >
generic_par_import        ::= Gen_Par_Imp type_name < cr > [generic_par_import]
generic_instantiation     ::= package_instantiation |
                              proc_instantiation |
                              func_instantiation
rename_use                ::= package_rename |
                              proc_rename |
                              func_rename
package_instantiation     ::= package_specification
                              Gen_Ins package_name < cr >
                              end_package_specification
proc_instantiation        ::= proc_specification
```

```
                          Gen_Ins unit_name < cr >
                          end_proc_specification
func_instantiation    ::= func_specification
                          Gen_Ins unit_name < cr >
                          end_func_specification
package_rename        ::= package_specification
                          Rena package_name < cr >
                          end_package_specification
proc_rename           ::= proc_specification
                          Rena unit_name < cr >
                          end_proc_specification
func_rename           ::= func_specification
                          Rena unit_name < cr >
subprogram_body       ::= proc_body_declaration |
                          func_body_declaration
proc_body_declaration ::= proc_body_name < cr > [formal_part]
                          [import_part] [basic_declaration]
                          body_detail
                          End_Of proc_body_name size_declaration
func_body_declaration ::= func_body_name < cr > [formal_part]
                          result_formal_part [import_part]
                          [basic_declaration] [body_detail]
                          End_Of func_body_name size_declaration
proc_body_name        ::= PROC_BODY unit_name
func_body_name        ::= FUNC_BODY unit_name
body_detail           ::= [usage_statements] [body_detail]
basic_declaration     ::= declarative_part [declarative_part]
declarative_part      ::= object_use |
                          type_use |
                          constant_use |
                          rename_use |
                          subprogram_declaration |
                          package_declaration |
                          task_declaration |
                          generic_declaration |
                          generic_instantiation |
                          body
object_use            ::= Var type_name < cr >
type_use              ::= Typ type_name < cr >
constant_use          ::= Con type_name < cr >
usage_statements      ::= call_statement |
                          use_statement
call_statement        ::= Call unit_name boolean < cr >
                          [act_par_list] [usage_statement]
                          End_Of Call < cr >
act_par_list          ::= General_Ass mode:type_name < cr >
                          [act_par_list]
use_statement         ::= Usage type_name < cr >
body                  ::= proper_body |
                          body_stub
body_stub             ::= Stub unit_declaration
unit_declaration      ::= subprogram_body_declaration |
                          package_body_declaration |
                          task_body_declaration
task_declaration      ::= task_specification
task_specification    ::= TASK_TYP task_name < cr >
                          [entry_declaration]
                          End_Of TASK_TYP task_name size_declaration
entry_declaration     ::= ENTRY_SPEC entry_name < cr >
                          [formal_part] [import_part]
                          End_Of ENTRY_SPEC entry_name size_declaration
task_body             ::= TASK_BODY task_name < cr >
                          [basic_declaration]
                          body_detail
                          End_Of TASK_BODY task_name size_declaration
```