Automated Improvement for Component Reuse

Muthu Ramachandran
School of Computing
The Headingley Campus
Leeds Metropolitan University
LEEDS, UK
m.ramachandran@leedsmet.ac.uk

Abstract

Software component reuse is the key to significant gains in productivity. However, the major problem is the lack of identifying and developing potentially reusable components. This paper concentrates on our approach to the development of reusable software components. A prototype tool has been developed, known as the Reuse Assessor and Improver System (RAIS) which can interactively identify, analyse, assess, and modify abstractions, attributes and architectures that support reuse. Practical and objective reuse guidelines are used to represent reuse knowledge and to do domain analysis. It takes existing components, provides systematic reuse assessment which is based on reuse advice and analysis, and produces components that are improved for reuse. Our work on guidelines has been extended to a large scale industrial application.

Keywords: Software reuse, component reuse, Development for reuse, Development with reuse, Reuse improvement, Reuse assessment

(Recebido para publicação em 6 de março de 2005 e aprovado em 20 de abril de 2005)

1. Introduction

Software component reuse is the key to significant gains in productivity. However, the major problem against the widespread introduction of reuse is the lack of identifying and developing potentially reusable components. We have clearly seen the difficulties that are faced when trying to reuse a component or a tool that is not designed for reuse. Therefore the objectives of this research are to explore the general area of **Development For Reuse** (DFR) and to investigate the possibility of automatically identifying, assessing and improving reusable domain abstractions, attributes and architectures. An objective of this process is to produce components that are potentially reusable as opposed to the normal practice of **Development With Reuse** (DWR) which has an objective of producing a product [1].

To achieve the production of reusable components we need to address the fundamental issue of what makes a component more reusable. Earlier studies have addressed this issue but do not go far from providing reusable guidelines [2-6]. Therefore, we took a more practical approach to address this issue by automating reuse guidelines for identifying, assessing, analysing and improving domain abstractions and attributes (*Domain analysis for reuse*) as well as identifying language features that affect component reusability (*Language analysis for reuse*). For example, certain languages (such as Java, C++, Ada95) support reuse explicitly. Engineers often cannot think about reuse when working on a market-driven project. In our approach we aim to integrate guidelines on language features and on domain analysis.

The notion of domain analysis has emerged from the well-known work conducted by Neighbors [7] on his pioneering project on the Draco system. Domain analysis aims to identify and design reusable components for a family of products. It also defines domain roles, process, and domain models and architecture. Existing work on domain analysis provides interesting guidelines, methods, and techniques on how to do domain analysis [8]. However, they fail to address, in detail, the issue of design for reuse. We took the existing work as a starting point for formulating reuse guidelines.

In our work, we have taken a more practical approach to domain analysis for the development of reusable software components by automating reuse guidelines. We also have defined the process of DFR, identifying domain abstractions & classification (domain-oriented reuse), language-oriented reuse, reuse assessment, and reuse improvement. Recently we have extended our work on guidelines into the design of reusable architectures for a large scale industrial application [9].

Our approach includes not only identifying abstractions and attributes but also assessing and adding these to improve components' reusability. A prototype has been developed, known as the Reuse Assessor and Improver System (RAIS). The major objective of this system is to demonstrate how well-

defined reuse guidelines can be used to automate the process of development of component reuse by providing support for language analysis and domain analysis. For example, this system takes an Ada component specification, assesses it through two analysis phases, estimates its reusability according to how well it satisfies a set of reuse guidelines and generates a component which is improved for reuse. Furthermore reuse improvement is done by performing various classes of structural and architectural transformations. Reuse assessment allows the identification of such structural abstractions early in the process. s.

In this context the system has demonstrated that it is possible to: (a) identify reusable abstractions, attributes and architectures effectively based on domain classification and reuse guidelines, (b) automate reuse guidelines which provide detailed advice on how to construct reusable components, (c) assist software engineers in the process of reuse assessment and improvement, (d) model reusable components based on templates (automated improvement), and (e) produce components that are potentially reusable.

In the following sections we discuss the process on development for reuse, reuse guidelines, the system that generates reusable components, an example, and an evaluation of the approach.

2. The Process of Development for Reuse

The main objective of this project is to provide a software system supporting the process of the development for reuse. In our work this process consists of various activities as shown in Figure 1:

Identify business needs - assess your existing system and application from the business point of view. What is the effort of building a new product? How much do we need to develop from scratch? How many components are you able to reuse? Justify your planned investment on reuse. Identify the application domain and its business/market needs. Define its boundary so that we can avoid producing components beyond the scope of the domain.

Identify & classify reusable abstractions, identify a list of components, frameworks, architecture, and utilities that share your business goals and can produce a high return-on-investment.

Formulate and classify reuse guidelines - produce reuse guidelines and classify them into domain-oriented reuse (i.e. guidelines on how to do domain analysis, guidelines on which abstraction has potential for reuse), design guidelines (guidelines on how design details/rationale can support reuse), architectural design guidelines, and language-oriented reuse (guidelines on language features).

Design components, make sure reuse engineers are familiar with reuse guidelines.

Assessment for reuse, allow other engineers' to conduct a reuse walkthrough or we can call it reuse inspection. Produce a detailed report following the inspection. It is interesting to see that reuse inspection is more structured and systematic since we have already formulated reuse rules.

Improvement for reuse, modify components based on the assessment report

Deliver potentially reusable components.

In this paper we concentrate mainly on two major activities, reuse assessment which is a process of assessing the reusability of a components against a set of well-defined guidelines, and reuse improvement which is a process of automatically modifying components structures and adding attributes that improve reusability.

We then identify reusable abstractions and classify them. The next step is to formulate practical reuse guidelines that can characterise reusable components effectively and precisely. The mechanism is based on taking the existing components, assessing these according to a set of guidelines, and then making suggestions on how the reusability of these components could be improved.

3. Reuse Guidelines as Knowledge Representation Technique

Probably there is no best and easy method of domain representation. Research is underway on how to do domain analysis, and on domain representation [8]. In our work, the approach taken is rule-based representation. Reuse guidelines are represented as rules. An example of the rule is:

IF abstract structure is complex AND
all operations are independent of the type of the structure element THEN
Component should be implemented as a generic package with the element type as a generic parameter;
END IF;

However, automating some of these guidelines breaches this rule. For example, one of our guidelines on defining the list of operations on object creation, termination, object inquiry, and state change, involves more than one interaction and transformations. Hence it breaches our single if-then rule and depends on applying domain knowledge for further transformations. This information is modelled using a component template and the reusability is assessed and improved by comparing the component with that template.

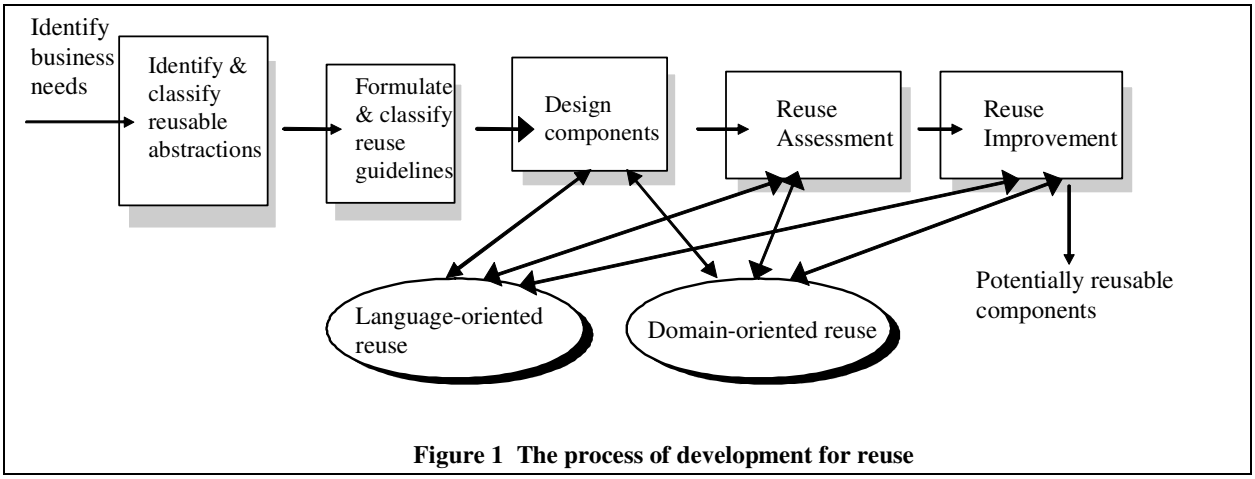

Figure 1 The process of development for reuse

Some of our guidelines are illustrated here:

1. *Design of abstract data types*. The notion of an abstract data type allows you to express real world entities of an application domain. It allows you to separate a specification from an internal representation of a structure (principle of information hiding). It means that we are able to specify an abstraction of a component in terms of its actual interface descriptions together which is useful to generalise that abstraction for reuse. It allows the designer to view a system at a more abstract level and to change the representation of ADS without affecting their use in other parts of the system.

One of our guidelines on ADS says,

• For all complex structures, provide two representations such as static and dynamic structures for each domain abstraction.

This guideline says, for each structure, provide two abstractions such as static which is represented using an array structure and dynamic which is represented using dynamic structure (access/pointer). This provides a choice and maximum flexibility for the reuser with improved reuse potential. For example, in Ada, we can design two packages for each structure implemented statically and dynamically. If an abstraction is to be represented in Ada then we can apply various Ada reuse guidelines. For example, one on the rationale for choosing private types. That is, choose limited private for complex and dynamic structures, and choose private type for static structures. However, the Ada library mechanism is inadequate in that it rises naming conflict when there are two library units with similar names which means that the implementation of similar components must have different names.

Another important guideline [4] on the design of abstract data structures emphasises the need for providing methods for a list of operations such as object creation, object termination, state change, state inquiry, and input and output. They have not considered operations on exceptions that deal with error conditions. We believe that the operations on exceptions and handling are significant for reusable and reliable components. In our work we have extended this guideline to include operations on exceptions handling.

Our extended guideline on ADS says,

The components should be provided with the following operations on ADS.
Creation
Termination
Conversion
State inquiry
State change
Input/ output representation, and
Exceptions

Creation involves both creating and initialising an object, termination is a means of making the object inaccessible for the remainder of its scope, conversion allows for the change of representation from one type to another, state inquiry functions allow the user to determine the state of the object and boundary conditions, state change functions allow modifying or changing the contents of the object, input/ output representations are primarily useful for debugging purposes, and exceptions deal

with error conditions and exception handling procedures. Each operation emphasises one or more functionality so that the services offered by the component are increased thus leading to improved reusability. Sometimes components which do not provide all these operations may well be reused. In such cases, the component has to be measured based on the degree of reusability.

2. *Other guidelines*. Our guidelines on the design of reusable static and dynamic structures, and on space management are essential, objective and realisable. Complete set of guidelines can be found in [1 and 9]. Some of our important domain guidelines are,

Always, define a constrained array structure to represent a component of static structure.

Always select dynamic object representation for all complex structures and hide detailed structural information.

- If the abstract structure is complex and all operations are independent of the type of the structure element then that component should be implemented as a generic package with the element type as a generic parameter.

Always provide a procedure to record the maximum size of the free list with a counter so that the user may increase or decrease the size of the free list. when decreasing the free list size, space in excess of the new size is returned to the system.

Always provide a procedure to release the free list, so that all space in the free list is returned to the system completely.

For each exception, provide an exception handler.

In the following section we will see how these guidelines can be implemented as a tool for automated improvement and advisory system which can take Ada code and provides an assessment and improvement for reuse.

4. The Reuse Assessor and Improver System (RAIS)

Reuse assessment is concerned with assessing the reuse potential of a component against reuse guidelines. Reuse improvement has the goal of transforming an assessed component into a component that is improved for reuse, based on language-oriented and domain-oriented reuse guidelines. This system takes an Ada component specification and estimates its reusability according to how well it satisfies a set of reuse guidelines and generates a component which is improved for reuse. The system produces assessment reports based on the percent of guidelines satisfied and interacts with the user for making further improvements.

A general model of the tool for systematic reuse assessment and improvement has been developed as shown in Figure 2.

The important features of this system are,

Identifying domain abstractions, attributes and architectures, and language attributes and structures that affect component reusability.

The integration of language knowledge (supporting language-oriented reusability) and domain knowledge (supporting domain-oriented reusability),

Providing reusability advice and analysis,

Assisting the reuse engineer in the process of assessing and improving his component for reuse.

RAIS considers a component specification rather than an implementation. However, this system can also generate implementation templates. We believe that reuse of specifications has definite advantages over reuse of implementations.

The RAIS system consists of a language analyser which is supported by built-in language knowledge and provides reusability analysis and advice, and a domain analyser which is supported by built-in domain knowledge and provides reusability analysis and advice.

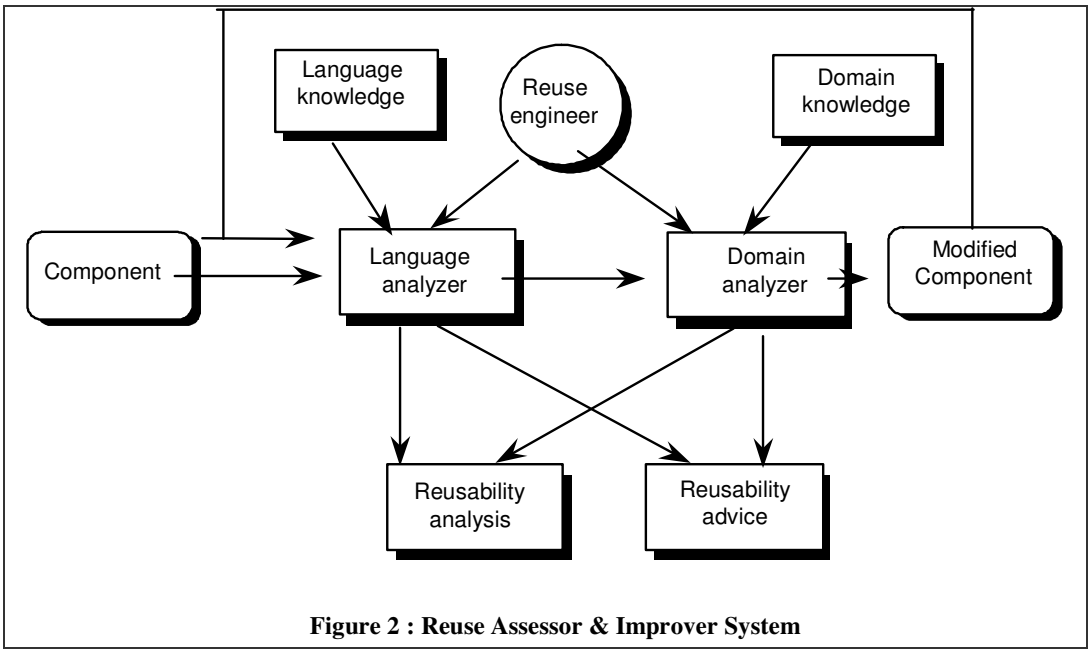

Figure 2 : Reuse Assessor & Improver System

An Ada component is firstly submitted to the language analyser which parses the component and applies the language-oriented guidelines to the code. Some of these guidelines require human input from the reuse engineer. RAIS predicts and records existing language constructs, and provides reuse advice and analysis. For example, the system can determine if the component processes arrays and if language attributes are used. However, it cannot automatically determine whether a component parameter refers to an array dimension and thus breaches the reuse guideline.

The language analyser assesses for reuse and changes the code after consulting the reuse engineer. The system interacts with the engineer to discover information that can't be determined automatically. The conclusion of this first pass is an estimate of how many guidelines are applicable to the component and how many of these have been breached. The report generator produces a report with all the information that has been extracted about that component and changes that have been made for reuse.

The second pass involves applying domain knowledge to the system. The component templates have been modelled representing static and dynamic structures. Their reusability is assessed by comparing the component against that template. Domain reuse improvement is done by adding methods automatically. Operation classes are identified by interaction with the reuse engineer. If some operations are found to be missing, skeleton implementations of these can be generated from the template for expansion to create a reusable component.

The support provided by the system ensures that the reuse engineer carries out a systematic analysis of the component according to the suggested guidelines. He or she need not be a domain expert. Again, an analysis is produced which allows the engineer to assess how much work is required to improve system reusability.

There are formulated reuse guidelines that emphasise the need for a packaging mechanism just like in Ada. Conceptually, packaging is a powerful mechanism for reuse. Some of these guidelines may only be possible with the Ada packaging mechanism such as private typing, the concept of specification which is independent of its body, and most importantly the concept of generics in order to achieve parameterisation. However, the approach and the methodology that are adopted by this system can easily be applied to any component. In this domain, RAIS uses the classification scheme in which each abstract data structure is classified into linear and non-linear structures and again these are classified into static, and dynamic structures.

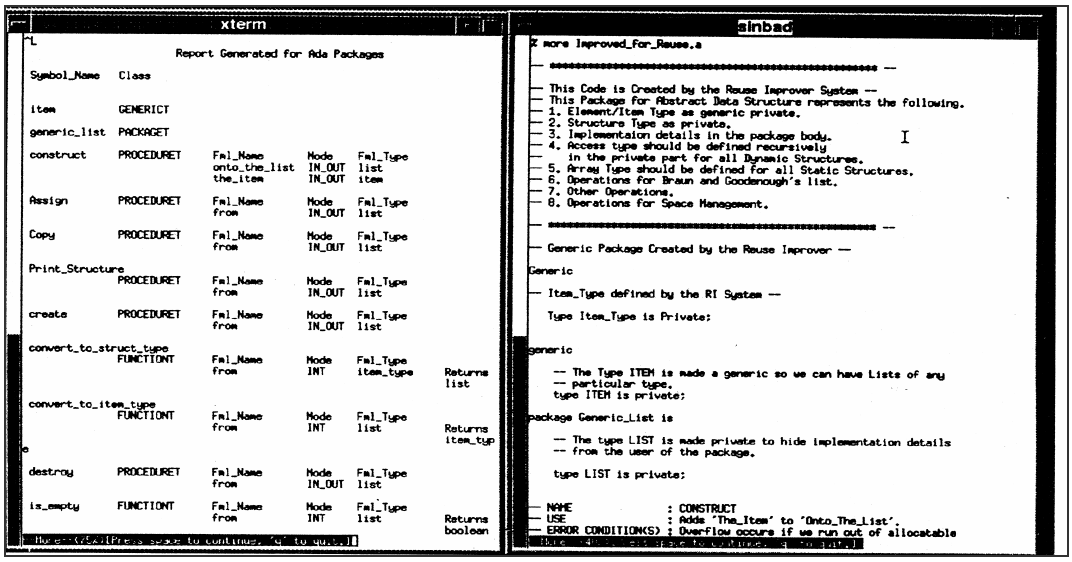

Figure 3 Assessment report and improved component

As well as this analysis, the system can also produce some reusability advice, generated from the guidelines, which is intended to assist the engineer in improving the reusability of the component. The knowledge of language and domain experts can be made available to the reuse engineer.

An ultimate objective is automatic reusability improvement where the system takes its own advice and some human guidance and modifies the component. A report and compilable code are produced. Clearly it is possible to use the language-oriented and domain-oriented guidelines to infer some code transformations which will improve reusability.

5. Reuse Assessment

Reuse assessment is a process of assessing the reuse potential of a component. It depends on the number of reuse guidelines that are satisfied by the component. RAIS predicts this and reports to the reuse engineer. RAIS measures the reusability strength of a component based on the percent of guidelines satisfied such as *weakly* (less than 50%), *strongly* (50-70%), *limitedly* (70-90%), *immediately reusable* (more than 90%) and also it takes into account the significance of a guideline (its importance for reuse).

For example, let us consider one of our domain guideline,

For all complex structures, the components should be implemented as a generic package with the element type as a generic parameter.

For instance, if a component of complex structure doesn't possess a generic package then the significance of this guideline becomes very important and therefore the system immediately reports to the reuse engineer that the component is weakly *reusable*. The system can make such structural modification automatically if the engineer decides to do so by responding to the dialogue.

In this way reuse assessment is being done by RAIS. The result of the assessment process is obviously arbitrary but it allows implementations to be compared, reuse improvements to be assessed, and it allows the reuse engineer to re-plan well before reusing components. The report generator produces the complete details of a component submitted to the systems in a tabular form which mainly consists of object name, its class, details of all the subprograms including the details of formal parameters and their class, and details of private types, etc. An example of a report is shown in a later section of this paper, see Figure 3.

6. Reuse Improvement

Reuse improvement is a stepwise process of improving a component for reuse through several transformations. Transformations can be simple, multiple, and cumulative. Because of the effort involved in this process, it has not been possible to implement for all the possible improvements. RAIS does most of the reuse improvements using reuse guidelines as domain rules and component templates. At present, RAIS can improve the component reusability by 50%.

Each abstract data structure is analysed and, by interaction with the user, the presence or absence of these operations is then identified. This information is modelled using a *component template* and the

reusability is assessed by comparing the component against that template. Operation classes are identified by interaction with the reuse engineer. If some operations are found to be missing, skeleton implementations of these are generated from the template for expansion to create a reusable component.

Two types of templates are created supporting reuse of architectures, one for static structures and another for dynamic structures. After reuse assessment, the designer is given all the information captured from his component (a report generator for Ada has been designed for this purpose). Finally, RAIS generates the component that is assessed and improved for reuse after several transformations.

The system has taken a pragmatic approach to domain analysis supporting development for reuse. Figure 3 shows the details of a report generated by the system after an initial analysis and assessment. Finally it generates the component which is improved for reuse.

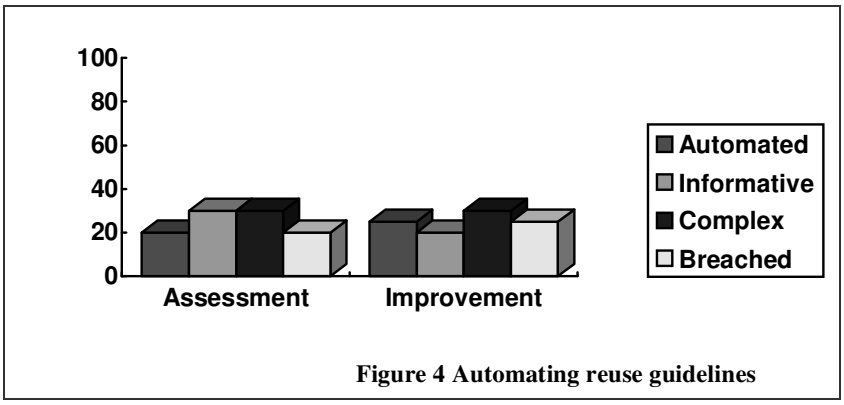

Figure 4 Automating reuse guidelines

7. Critical Evaluation

Existing approaches have not explored the issues of development for reuse and others have considered this as a management problem. In this context, our work has explored one of the major technical problems and the system has demonstrated that it is possible to assess and improve components reusability automatically. This work has also demonstrated that it is possible to formulate object and practical reuse guidelines that can assist and advise software engineers on how to construct components that are potentially reusable. This is one of the major practical steps taken in this work. Figure 4 illustrates how guidelines are classified and how many are automated.

RAIS has also demonstrated that the integration of language knowledge and the application domain knowledge is possible when modelling components for reuse. Therefore we feel that the various steps proposed for the process of development for reuse are important, practical and can be considered along with or before the normal software development process.

The system has also proved perhaps to a limited extent that it is possible to design for the highest form of reuse which is the reuse of components and architectures. The system models components effectively based on the templates for reuse of component architectures that are static and dynamic. It is not quite clear for example on what is probably the best technique for domain representation, what should be considered as a domain, and so on. In this context we might feel that the application domain chosen is perhaps inadequate in the commercial sense. However we believe that it is possible to extend the approach described here to other application domains, languages, and tools.

It has not been possible to automate all the guidelines that are formulated but it should be possible in a long-term project. The system does perhaps a limited number of domain-oriented reuse improvements. We believe that it is also possible to extend the approach described here to higher levels of reuse such as requirements definition and specification.

8. Conclusions

The objectives of this project were to explore the general area of development for reuse and to investigate the possibility of automatically assessing the reusability of a software component and modifying that component to improve its reusability. In this context, the system has demonstrated that it is possible to identify, assess and improve components' reusability automatically based on domain knowledge and language knowledge.

In addition to these, more interesting results have evolved from this research, reusing generic component templates and generic architectures. Further work is needed to enhance the functionalities of RAIS. We believe that it is possible to extend the approach described here to other domains, languages and tools. Our work on reuse guidelines has been applied to a large-scale industrial application [9].

9. Acknowledgement

The author wishes to thank Prof Ian Sommerville for his support during this work at Lancaster University, Lancaster, UK.

10. References

[1] Sommerville, I. and Ramachandran, M. (1991), Reuse Assessment, First International Workshop on Software Reuse, Dortmund, Germany, July.

[2] Hooper, J. W. and Chester, R. O. (1991). Software Reuse: Guidelines and Methods, Plenum Press.

[3] Gautier, R.J. and Wallis, P.J.L. (Editors) (1990), Software Reuse with Ada, Peter Peregrinus Ltd for IEE/BCS.

[4] Braun, C.L. and Goodenough, J.B. (1985), Ada Reusability Guidelines, Report 3285-2-208/2, USAF.

[5] Booch, G. (1987), Software Components with Ada, Benjamin/Cummings.

[6] Dennis, R.J.St. (1987), Reusable Ada(R) software guidelines, Proc. of the 12th annual Hawaii International conference on system sciences, pp.513-520.

[7] Neighbors, J.M. (1984), The Draco Approach to constructing Software from reusable components, IEEE Trans. on Software Engineering, vol.SE-10, No.5, pp.564-574, September.

[8] Prieto-Diaz, R and Arango, G (ed) (1991), Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

[9] Ramachandran , M and Fleischer, W. (1996). Design for large scale reuse: an industrial case study, Proceedings of the 4th Intl. Conf. on Software Reuse, IEEE CS press, Orlando, Florida, USA.

[10] Tracz, W. (1991), Reuse through parameterization, ACM SIGSOFT Software Eng.Notes.