Dr. Huiming Yu
Department of Computer Science
College of Engineering
NC A&T State University
Greensboro, NC 27411
Fax:(336)334-7244
Phone: (336)334-7245
Email: cshmyu@ncat.edu

April 6, 1999

Dear Mr. Taft,

The included paper "An Approach for Extracting Objects in Ada 83 Programs" is for The SigAda ‘99. The research result does not been submitted elsewhere. If you have any questions, please do not hesitate to contact me at (336)334-7245.

Thank you for your time and consideration.

Sincerely,

Huiming Yu

An Approach for Extracting Objects in Ada 83 Programs

Yuming Zhou Baowen Xu
Department of Computer Science & Engineering
Southeast University
Nanjing 210096, P. R. China
{adalab, bwxu}@seu.edu.cn

Huming Yu
Department of Computer Science
North Carolina A&T State University
Greensboro, NC27411
cshmyu@ncat.edu

Abstract

In this paper a new approach to extract objects, from legacy systems, that are written in Ada83 is presented. Different from existing global-based object identification and type-based object identification approaches, this approach uses features of modules to classify objects. Several module cohesion metrics are proposed, inheritance relations among objects are analyzed, and an object extraction algorithm is developed.

Keywords
Object identification, Module cohesion, Inheritance, Ada 83, Ada 95

1. Introduction

There currently exist a large number of legacy systems which are written in Ada 83. These systems are difficult to understand and maintain because Ada 83 is an object-based programming language. Redeveloping reliable and equivalent object-oriented systems in Ada 95 to replace these legacy systems may need several years with high cost. A quicker solution is to reengineer these legacy systems using object-oriented features of Ada 95. The authors propose an algorithm that transforms serving tasks written in Ada 83 into protected objects in Ada 95 to make system maintenance easier and improve system performance [6].

An object is a collection of attributes and methods. The key of object identification in programs written in conventional languages is to group related types, data items and subprograms into one object through analysis of program behaviors. Different approaches of extracting objects from legacy systems written in conventional procedural-oriented languages have been developed. In the paper [1] Liu and Widle propose a global-based object identification approach and a type-based object identification approach. These approaches map global variables or types to objects' attributes, and map subprograms referencing these global variables to objects' methods. In the paper [3] Panos proposes a receiver-based method through which only modified variables are used to classify subprograms. A common limitation of those approaches is that data and subprograms in legacy programs are simply grouped by types or global variables not by module features, and extracted objects may not be real objects in application domains. Moreover, most object identification approaches do not extract the potential inheritance relations among objects.

We have developed a new object extraction approach that includes four steps. First, a graph representing the links that exist among types and subprograms is generated by the analysis of program behaviors. Second, module cohesion analysis is performed on the graph to group related types and subprograms into candidate objects. Third, potential inheritance relations among objects are analyzed and extracted. Finally, extracted objects are implemented using the object-oriented mechanisms of Ada 95. This paper is organized as follows. In section 2 a graph representing the relations of types and subprograms is presented, in section 3 analysis of module cohesion is given, and in section 4 an object identification algorithm is presented. Concluding remarks are given in section 5.

2. Relations among Types and Subprograms

In the type-based object identification approach, a topological ordering of types is defined and used to classify subprograms [1]. However, this type-ordering scheme assumes that all types in a program are related in terms of the component relationships (Type t1 is a component type of type t2 only if t1 is used to define t2). In order to handle this case that some types are not related in terms of the component relationship, Ogando defines the relative complexity of types, and then those types with high complexity are used to classify subprograms into a type-based object [4]. We are convinced that different types play different roles in object identification, therefore, all types contribute to the classification of subprograms. Referring to the definition of the relative complexity of types in paper [4], the complexity of a type t in Ada 83 programs is calculated as follows.

$$C_{t,\,l} = \begin{cases} v + l & \text{if t is an elementary type} \\ Complex * C_{t',\,l+1} & \text{if t is a pointer (Complex > 1) to type t'} \\ C_{t',\,l} & \text{if t is a subtype of type t'} \\ C_{t',\,l+1} & \text{if t is a derived type of type t'} \\ f(C_{t1,l+1}, C_{t2,\,l+1}, \ldots, C_{tn,\,l+1}) & \text{if t is a composite type composed of component type } t_i \end{cases}$$

where:

- l is the nested depth of type t relevant to its root type,
- v is the complexity of an elementary type,
- Complex represents the degree that the pointer's complexity affects the complexity of the types in Ada 83 language,
- f is a function to represent complexity of its component type t_i. It is defined as $\Sigma C_{ti,\,l+1}$ if f is a record type composed of component type t_i. If t is an array type, f is defined as dim $*$ 2 $*$ $C_{t',\,l+1}$, where dim is the dimension of array t and t' is the type of the array element.

It should be noted that the complexity of an array type is closely related to its dimension rather than its element number, and that an array type of one dimension is more complex than a record type composed of only one component type. Therefore, the complexity of an array type t is defined as dim $*$ 2 $*$ $C_{t',\,l+1}$. In this paper, for simplicity the complexity of generic types in Ada 83 is not considered. The following example illustrates the use of the complexity function $C_{t,\,l}$ of type t.

```
  type GENDER is (FEMALE, MALE);
```

```
type HEIGHT_CM is range 0..300;
type PERSON is record
   SEX      : GENDER;
   NAME     : STRING;
   HEIGHT   : HEIGHT_CM := 0;
end record;

type BUS is record
    DRIVE   : PERSON;
    SEATS   : POSTITIVE;
end record;
```

The complexity of BUS is calculated as follows.

$$\begin{aligned} C_{BUS,\,0} &= C_{PERSON,\,1} + C_{POSITION,\,1} \\ &= C_{STRING,\,2} + C_{HEIGHT\text{-}CM,\,2} + C_{GENDER,\,2} + (1 + v) \\ &= (2 + v) + (2 + v) + (2 + v) + (1 + v) \\ &= 7 + 4 * v \end{aligned}$$

A subprogram may be considered as a black box that performs the transformation from its formal parameters to a return value. In Ada 83, subprograms can access global variables directly. Therefore, the function of a subprogram is related to the types of its formal parameter(s), its return value and global variables referenced by the subprogram. We use a graph to represent the relations that exist among types and subprograms, and call it a ST (Subprogram-Type) graph. Assume that S_{Subp} is the set of subprograms and S_{type} is the set of types, then the ST graph for this program can be formally defined as follows:

Definition: ST graph
A graph G = (N, E) is a ST graph if $N = S_{Subp} \cup S_{type}$ and $E=\{(p, t) \mid p \in S_{subp} \wedge t \in S_{type} \wedge t \in T$, where T is the set of types of global variables referenced by p, the return value and formal parameters of subprogram p$\}$, and $C_{t,\,0}$ is the weight of edge (p, t).

Obviously, the nodes of a ST graph fall into two categories: subprogram nodes and type nodes. For each subprogram node p and type node t of the ST graph, we define $S_{p\text{-}t(p)}$ and $S_{t\text{-}p(t)}$ as:
$S_{p\text{-}t(p)} = \{ t \mid t \in S_{type} \wedge (p, t) \in E \}$
$S_{t\text{-}p(t)} = \{ p \mid p \in S_{Subp} \wedge (p, t) \in E \}$

Actually, a ST graph is a bipartite graph. In Ada 83 programs, the links existing among subprograms and types are represented by the graph. The simplest object identification approach is to find all isolated sub-graphs of the ST graph in legacy systems. Each sub-graph is a candidate module to be an object. However, this method does not produce satisfactory results. Sometimes it may group subprograms, which belong to logically different objects, into one object. For example, assume there is a subprogram `Push_Stack(S: in out Stack_Type; Elem: in Elem_Type)` and a subprogram `Enter_Queue(Q: in out Queue_Type; Elem: in Elem_Type)` in a program. Both subprograms are on the same sub-graph of the ST graph because of the common type `Elem_Type`. According to the object identification method that each isolated sub-graph is to be a candidate object, `Push_Stack` and `Enter_Queue` will be grouped into the same object. Sometimes attributes belonging to logically different objects may also be grouped into the same object. For example, if there exists a subprogram `Init(S: in out Stack_Type; Q: in out Queue_Type)` that initializes Stack object and Queue object, then type `Stack_Type` and `Queue_Type` exist in the same sub-graph of the ST graph because of the subprogram `Init`. The object identification method by finding all isolated sub-graphs will group these types to the same object. The object identification approach presented in this paper based on module cohesion metrics overcomes its limitations.

3. Module Cohesion Analysis

In OO systems, a class is a module that is a collection of related types and operations. Module Cohesion refers to the degree of interdependence among components of a module. Module Coupling refers to the degree of dependence among modules. A good software design should obey the principle of high cohesion and low coupling. In the object identification of legacy systems, high cohesion objects can be extracted, therefore the performance and maintenance of the equivalent systems transformed by reengineering of legacy systems are improved. Although a number of object identification approaches have been proposed, a new approach that considers module cohesion and coupling is necessary. In this section, module cohesion metrics are proposed to quantify the level of cohesion of objects extracted from Ada 83 programs.

3.1 Module Cohesion Metrics

Assume a function named MT_{int}, which is applied to a ST graph of a module, gives the intersection types of the types of global variables referenced by subprograms, formal parameters and the return value of all subprograms of the module, such that

$$MT_{int}(G) = \cap_{p \in Ssubp} S_{p\text{-}t}(p)$$

Based on the definition of MT_{int} of a module, module cohesion metrics is developed. It is described as follows.

Definition: Module tightness
Module tightness Tightness(G) of a module is defined as a division of the sum of complexity degrees of types in $MT_{int}(G)$ by the sum of complexity degrees of types in S_{type}, where G is the ST graph of the module.

$$\text{Tightness}(G) = \frac{\Sigma_{t \in MTint}(G)\ (C_{t,0} * |S_{t\text{-}p(t)}|)}{\Sigma_{t \in Stype}(G)\ (C_{t,0} * |S_{t\text{-}p(t)}|)}$$

High module tightness reflects that the degree of intersection among the types of global variables referenced by subprograms, formal parameters and return values of subprograms of a module is high, or that the sum of complexity degrees of types that intersect is high. High module tightness certainly means that the degree of interdependence among subprograms of the module is high, i.e., cohesion of the module is high. Obviously, when $MT_{int}(G) = S_{type}$, module tightness has maximum value 1.

Definition: Module overlap
Module overlap Overlap(G) of a module is defined as the average value of the sum of complexity degrees of types in $MT_{int}(G)$ divided by the sum of complexity degrees of types in $S_{p-t(p)}$ for each subprogram p, where G is the ST graph of the module.

$$\text{Overlap}(G) = (1/|S_{Subp}|) * \Sigma_{p \in Ssubp} \frac{\Sigma_{t \in MTint(G)} C_{t,0}}{\Sigma_{t \in sp-t(p)} C_{t,0}}$$

High overlap means that the degree of intersection among types in $MT_{int}(G)$ and $S_{p-t}(p)$ for each subprogram of the module is high. Therefore, high module overlap also shows a high cohesion.

Both module tightness and module overlap provide measurements to evaluate the degree of interdependence among the components of a module. Based on these metrics, we can provide a precise guidance to extract objects from legacy systems that are written in Ada 83.

In order to extract objects with high cohesion from legacy systems we need to judge which subprograms should be grouped into the same object. Effects on cohesion from adding a subprogram to or deleting a subprogram from a module are discussed in the following two subsections.

3.2 Effects on Cohesion from Adding a Subprogram to a Module

That a module has a ST graph G will be named as original module. Add a subprogram p' to the original module, the new module will have a corresponding ST graph G' and it is named as result module. Based on the definitions of MT_{int}, module tightness and module overlap, the following lemmas are easy to be deduced.

Lemma 1

$$\frac{\Sigma_{t \in MTint(G) \cap Sp-t(p')} C_{t,0} - \Sigma_{t \in MTint(G) - (MTint(G) \cap Sp-t(p'))}(C_{t,0} * (|S_{p-t}(p')|)}{\Sigma_{t \in Sp-t(p')} C_{t,0}} >=$$

$$\text{Tightness}(G)$$

$$\Leftrightarrow \text{Tightness}(G') \geq \text{Tightness}(G)$$

Lemma 2

$$\frac{\Sigma_{t \in MTint(G')} C_{t,0}}{\Sigma_{t \in Sp-t(p')} C_{t,0}} \geq (|S_{subp}| + 1) * \text{Overlap}(G) - \Sigma_{p \in Ssubp} \frac{\Sigma_{t \in MTint(G')} C_{t,0}}{\Sigma_{t \in Sp-t(p')} C_{t,0}}$$

$$\Leftrightarrow \text{Overlap}(G') \geq \text{Overlap}(G)$$

When a subprogram p' is added to the original module, Lemma 1 shows us that the module tightness of the result module increases iff the value of the sum of complexity degrees of the types in MT_{int} from the result module to the original module divided by the sum of complexity degrees of the types in $S_{p-t(p')}$ is larger than the module tightness of original module. By Lemma 2, module overlap of result module may go up or down, depending on the function relation of MT_{int} of the result module, $|S_{subp}|$ and the module overlap of the original module.

Assume $MT_{int}(G) \subseteq S_{p-t}(p')$, we can deduce corollary 1 from Lemma 1 and Lemma 2.

Corollary 1

$$\text{(1) } Overlap(G) \le \frac{\Sigma_{t \in MTint(G)} C_{t,0}}{\Sigma_{t \in Sp\text{-}t(p')} C_{t,0}} \Rightarrow Overlap(G') \ge Overlap(G)$$

$$\text{(2) } Tightness(G) \le \frac{\Sigma_{t \in MTint(G)} C_{t,0}}{\Sigma_{t \in Sp\text{-}t(p')} C_{t,0}} \Rightarrow Tightness(G') \ge Tightness(G)$$

Obviously, $Overlap(G') \ge Overlap(G)$ and $Tightness(G') \ge Tightness(G)$ if $MT_{int}(G) = S_{p-t}(p')$.

3.3 Effects on Cohesion from Deleting a Subprogram from a Module

Deleting a subprogram from a module has a reverse effect on cohesion compared to adding a subprogram to a module. Assume p' is the subprogram deleted from a module. Lemma 3 and 4 can be easily deduced from the definitions of module tightness, module overlap and MT_{int}.

Lemma 3

$$\frac{\Sigma_{t \in MTint(G)} C_{t,0} - \Sigma_{t \in MTint(G') - MTint(G)} (C_{t,0} * (|S_{p-t}(p')|)}{\Sigma_{t \in Sp\text{-}t(p')} C_{t,0}} \le Tightness(G)$$

$$\Leftrightarrow Tightness(G') \ge Tightness(G)$$

Lemma 4

$$\frac{\Sigma_{t \in MTint(G')} C_{t,0}}{\Sigma_{t \in Sp\text{-}t(p')} C_{t,0}} \le \Sigma_{p \in Ssubp} \frac{\Sigma_{t \in MTint(G')} C_{t,0}}{\Sigma_{t \in Sp\text{-}t(p')} C_{t,0}} - (|S_{subp}| - 1) * Overlap(G)$$

$$\Leftrightarrow Overlap(G') \ge Overlap(G)$$

Both lemma 3 and 4 provide an evaluation of the effect on cohesion by deleting subprograms from a module, and thereby show which subprograms should be deleted from a module in order to increase module cohesion.

Assume $MT_{int}(G) = MT_{int}(G')$, we can deduce corollary 2 based on Lemma 3 and 4.

Corollary 2

(1) $\text{Overlap}(G) \geq \dfrac{\Sigma_{t \in \text{MTint}(G)} C_{t,\,0}}{\Sigma_{t \in \text{Sp-t}(p')} C_{t,\,0}} \Rightarrow \text{Overlap}(G') \geq \text{Overlap}(G)$

(2) $\text{Tightness}(G) \geq \dfrac{\Sigma_{t \in \text{MTint}(G)} C_{t,\,0}}{\Sigma_{t \in \text{Sp-t}(p')} C_{t,\,0}} \Rightarrow \text{Tightness}(G') \geq \text{Tightness}(G)$

4. Inheritance Analysis among Objects

Different from other procedure-oriented languages, Ada 83 provides encapsulation and abstract data types through the facilities of packages and private types. The set of operations defined for a subtype of a given type includes the operations defined for the type. A derived type declaration defines a new type whose characteristics are derived from a parent type. For each basic operation of the parent type, there is a corresponding basic operation of the derived type. For each derivable subprogram of the parent type, there is a corresponding derived subprogram for the derived type [5]. Therefore, a subtype implicitly inherits the operations of its base type and a derived type implicitly inherits the derivable operations of its parent type. In addition, we may define new operations on subtypes and derived types. In Ada, type definition is separated from program modularization. The definitions of types and operations of an object are encapsulated by means of packages. Through analysis of the subtype and derived type relation among object types, we can extract inheritance relations among objects. The following is an example.

```
package Person_Package is
 type PERSON(Sex: GENDER) is
  record
   Birth: DATE;
   Sex:  GENDER;
   Father:  PERSON_NAME;
   Mother:  PERSON_NAME;

   case Sex is
    when MALE =>
     Wife: PERSON_NAME;
    when FEMALE =>
     Husband: PERSON_NAME;
   end case;
 end record

 procedure GetBirth
  (In_Person: in PERSON;
   rth_Day : out GENDER);
 …
end Person_Package;
```

Figure 1

```
with Person_Package;
package Man_Package is
  subtype MAN is PERSON(MALE);
  procedure GetWifeName(In_Man : in MAN;
                        WifeName : out PERSON_NAME);
  …
end Man_Package;
```

Figure 2

```
with Person_Package;
package Woman_Package is
  type WOMAN is new PERSON(FEMALE);
  procedure GetHusbandName(In_Woman: in WOMEN;
                           HusbandName: out PERSON_NAME);
  …
end Woman_Package;
```

Figure 3

Type MAN is a subtype of type PERSON and type WOMAN is a derived type of type PERSON. Both types MAN and WOMAN in effect “inherit” the operations of the type PERSON. In our method, we identify this kind of “inheritance” among objects through identifying the subtype or derived type relations among object types. This kind of inheritance allows operations of an object to be inherited by another object but doesn’t allow new attributes to be added. Another kind of “inheritance” which supports attributes addition of objects should also be identified. Figure 4 is a simple example.

```
with Person_Package;
package Man_Package is
  type MAN is record
    Self: PERSON_PACKAGE.PERSON(MALE);
     …
  end record;
  procedure GetWifeName(In_Man: in MAN;
                        WifeName : out PERSON_NAME);
  …
end Man_Package;
```

Figure 4

Now, type MAN doesn’t inherit operations of the PERSON type. However, Self is a field of MAN and therefore the MAN type can indirectly call operations of PERSON by Self. Semantically, it is possible that there exist inheritance relations among those types. Our new approach identifies these two kinds of “inheritance” among objects. Then they may be transformed using OO mechanisms of Ada 95.

In order to identify the possible “inheritance” relations among objects in a program, we define a sub-graph corresponding to a given type node on the ST graph.

Definition For a type node t on a ST graph G = <N, E> of a module, construct a sub-graph Gt = $<N_t, E_t>$ of the ST graph G such that

(1) $N_t = PP_t \cup TT_t$, where $PP_t = S_{t\text{-}p}(t)$ and $TT_t = \cup\, p \in S_{t\text{-}p}(t)\, S_{p\text{-}t}(p)$.

(2) $\forall e = <pn, tn> \in E$, if $pn, tn \in N_t$ then $e \in E_t$.

For a type t, the sub-graph G_t is generated by clustering together all subprograms related to t and all types related to those subprograms. Among all types in TT_t, the type t is considered to be the object type of the module corresponding to the ST graph G. Therefore, we use type t to identify the possible inheritance relation among objects according to the following rules.

Rule 1 For sub-graph $G_{t1} = <N_{t1}, E_{t1}>$ and $G_{t2} = <N_{t2}, E_{t2}>$ of a ST graph G, if type t2 is a subtype or a derived type of type t1, then the object corresponding to sub-graph G_{t2} inherits from the object corresponding to sub-graph G_{t1}.

Rule 2 For sub-graph $G_{t1} = <N_{t1}, E_{t1}>$ and $G_{t2} = <N_{t2}, E_{t2}>$ of a ST graph G, if t1 is the type of one component of type t2, then the object corresponding to sub-graph G_{t2} inherits from the object corresponding to sub-graph G_{t1}.

5. Object Extraction Algorithm

There exist coincidental connections and spurious connections on the ST graph of a module. Coincidental connections are caused by subprograms that implement several functions, and each function logically belongs to a different object. Spurious connections are caused by subprograms that implement specific system operations by directly accessing the attributes of more than one object. Splitting a subprogram that causes coincidental connections to different subprograms, each subprogram logically implementing a function, can eliminate coincident connections.

We have developed an algorithm to extract objects with high cohesion from legacy Ada 83 systems. In this algorithm a step value STEP is used to quantify the informal concept of high cohesion. STEP can be defined based on statistics. For a legacy system written in Ada 83, we construct the ST graph and calculate module cohesion of sub-graph G_t for each type node t. Each subprogram in the intersection of subprogram nodes of these sub-graphs determines which module it should belong to. These sub-graphs that have common type nodes determine if they should be merged. The input to this algorithm is a legacy Ada 83 program and the output is a set of objects with high cohesion.

Algorithm

(1)Initializes STEP to a step value and set list OverStep empty.

(2)For each type t of the ST graph, compute module cohesion of the corresponding sub-graph G_t. If its module cohesion is larger than STEP, add G_t to list OverStep.

(3)$\forall G_{t1}, G_{t2} \in$ OverStep, where t1 and t2 are in the intersection of type nodes of G_{t1} and G_{t2}.

a) If module cohesion of the sub-graph G_t generated by merging G_{t1} and G_{t2} is larger than STEP, then G_{t1} and G_{t2} are removed from OverStep and G_t is added to OverStep. Regenerate a new ST graph, with t1 and t2 being merged into a new type t.

b) If module cohesion of the sub-graph G_t generated by merging G_{t1} and G_{t2} is less

than STEP, split those subprograms that cause coincidental connections and remove those subprograms that cause spurious connections from both G_{t1} and G_{t2}.
(4) Go to (2), until when $\forall G_{t1}, G_{t2} \in$ OverStep, either type t1 or t2 is in the intersection of type nodes of G_{t1} and G_{t2} or neither is in.
(5) $\forall G_{t1}, G_{t2} \in$ OverStep, assume the intersection of their corresponding subprogram nodes is Set, we use G'_{t1}, G'_{t2} to represent the sub-graphs generated by removing nodes in Set and corresponding edges from G_{t1}, G_{t2} respectively. $\forall p \in$ Set, p is added to G'_{t1} if one of the following three conditions holds:
a) the cohesion of G'_{t1} increases and the cohesion of G'_{t2} decreases when p is added,
b) the increment of cohesion G'_{t1} is larger than the increment of cohesion G'_{t2} when p is added,
c) the decrement of cohesion G'_{t1} is larger than the decrement of cohesion G'_{t2} when p is added.
Remove G_{t1}, G_{t2} from OverStep and add G'_{t1}, G'_{t2} to OverStep.
(6) Go to (5), until when $\forall G_{t1}, G_{t2} \in$ OverStep, the intersection of the subprogram nodes in G_{t1} and G_{t2} is empty.
(7) Each sub-graph $G_t \in$ OverStep is encapsulated into an object. Extract the inheritance relations among objects in terms of rule 1 and 2 presented in the last section.

6. Conclusions

As module cohesion and coupling are the fundamental aspects of software design, there are a few object identification methods extracting objects from procedure-oriented systems using module features, we develop a new approach that is based on module cohesion for extracting objects from legacy Ada 83 systems. This approach focuses on the extraction of objects with high cohesion. Several module cohesion metrics are presented and effects on module cohesion from adding subprograms to or deleting subprograms from a module are analyzed. Considering the features of types in Ada 83, we analyze the relations among objects and develop rules on how to extract inheritance relations among objects. Traditional object identification methods based on types or global variables may group subprograms or types logically belonging to different objects into the same object and they don't consider the extraction of inheritance relation among objects. Our solution overcomes these limitations. We are currently extending our approach to couple metrics.

7. Acknowledgements

The authors gratefully acknowledge the contributions of Xiaoyu Zhou, Banqing Li, Lujun Zhang and Guilan Dai in the preparation of this paper.

References

[1] Liu, S. S. and Wilde, N. (1990), " Identifying objects in a conventional procedural language ", in Proceedings of the Conferences on Software Maintenance, 266-271, November 1990.
[2] G. Canfora, A. Cimitile and M. Munro, C. J. Taylor, "Extracting Abstract Data Types from C Programs: A Case Study(," 1063-6773/93 IEEE.
[3] Panose. Livadas and Theodore Johnson, " A New Approach to Finding Objects in Programs," Software Maintenance: Research and Practice, vol. 6. 249-260(1994).

[4] Roger M. Ogando, Stephen S. Yau , Sying S. Liu and Norman Wilde, “An Object Finder for Program Structure Understanding in Software Maintenance,” Software Maintenance: Research And Practice, Vol . 6, 261-283 (1994).
[5] ANSI/MIL-STD-1815A-1983(ISO 8652-1987), Reference Manual for the Ada Programming Language, 1983
[6] Bangqing Li, Baowen Xu, and Huiming Yu, “Transforming Ada Severing Tasks Into Protected Objects”, Proceedings Of Acm Sigada Annual International Conference, November 8-12, 1998.