Migration of Procedural Systems to Network-Centric Platforms *

Prashant Patil[1] Ying Zou[1] Kostas Kontogiannis[1] John Mylopoulos[2]

University of Waterloo[1]
Dept. of Electrical & Computer Engineering

University of Toronto[2]
Dept. of Computer Science

Abstract

Technologies developed over the past few years such as CORBA, Java and the Web, have made it easier to build and deploy distributed object applications. These technologies have also made a visible impact on legacy software system evolution.

This paper focuses on the methods for re-engineering procedural systems into new Network-Centric platforms. The first step of this re-engineering method is to migrate a legacy system into an object oriented architecture. The extraction of the object oriented architecture is based on global data type analysis and an evidence model that allows for alternative target designs to be evaluated and ranked. Once a target design has been extracted, C++ code is generated for the re-engineered system. The second step is to wrap the components of the new object oriented system with interfaces, so that they can be made available through a network centric workbench such as CORBA. Automatically generated IDL interfaces, and CORBA compliant wrapper classes allow for the migration of the new code to a distributed Network-Centric environment. A prototype tool has been built and in this paper we discuss how the tool and these re-engineering techniques can be applied to migrate three medium size C systems.

*This work was funded by IBM Canada Ltd. through the Toronto Laboratory, Center for Advanced Studies; also by the Natural Sciences and Engineering Research Council of Canada. Contact persons at IBM CAS for this work are Bill O'Farrel (billo@ca.ibm.com) and Stephen Perelgut (perelgut@ca.ibm.com).

1 Introduction

Over the past few years, the explosive growth and acceptance of technologies such as the Internet, CORBA, Java and the Web have attracted attention away from client-server architectures and have shifted it towards distributed object ones.

In this paper, we propose re-engineering techniques and tools that utilize distributed object technology to re-engineer procedural legacy software systems to new Network-Centric Platforms.

This paper presents techniques for incrementally migrating a procedural system into a distributed, object-oriented architecture. The first step of the migration process extracts an object model from legacy source code. This step can be quite tedious, but it has been partially automated by a tool that helps the user in developing an object-oriented design for the new system. Object extraction from legacy code and method attachment is guided by the tool and is based on an evidence model. The object model that is produced actually goes through several iterations and evolves according to user's knowledge. Once object oriented components have been identified and corresponding C++ code has been generated, these objects can be made available through wrapper classes and CORBA compliant IDL specifications. The wrapper classes and IDL specifications can be generated automatically from the generated C++ code using our tool.

This paper is organized as follows. Section 2 discusses the related work, section 3 presents the methodology adopted for object identification. In section 4, we have described method identification techniques and conflict resolution strategy. Section 5, gives details about how the class-method conflicts are resolved. In section 6 source code generation process is described and the refinement of evolved object model is described in section 7. Section 8 discusses the wrapping of objects and making them available through CORBA interface. In section 9, experimentation involved in this work is mentioned and section 10 lists concluding remarks.

2 Related Work

The migration of procedural legacy systems to object oriented systems has been addressed by a number of research groups in the software engineering community. Overall, these research efforts fall under two main categories. The first category includes techniques which focus on the identification of objects and abstract data types (ADT).

In [11] an interactive system that finds candidate ADTs and object instances in procedural systems is presented. The basis of the analysis is the program's Abstract Syntax Tree (AST). A *field reference* graph provides information on the dependencies and uses of different structure (or record) fields by various functions or procedures of the subject system. [19] describes a system that allows for the identification of objects in procedural systems using domain knowledge and user interaction. An object model is extracted from the source code and is compared to an independently developed object model of the underlying system. [17] proposes clustering techniques to identify ADTs from existing procedures. The technique is based on attaching data types to existing procedures. A collection including a data type with its associated operations is then identified as an ADT. In [8] an object identification method based on the global data types and on the use of formal parameters is proposed. Finally, [15] presents a system for the identification of objects in COBOL programs. The approach uses slicing techniques to identify all elementary operations which change the state of a data type.

The second research direction focuses on re-engineering and restructuring an existing application in an object oriented way. This includes source code transformations, clustering, wrapping and integration with other systems. [13] discusses a re-engineering tool that allows for the transformation of redundant, duplicated and similar data and processes to classes and methods. Likewise, [3] proposes a tool that supports the decomposition of large legacy system. The technique can be used to re-engineer a system into a client-server platform. In [16] a system for re-engineering COBOL legacy applications to distributed object oriented environments are presented. The system deals with the problem of encapsulating legacy system components in object wrappers. Other interesting work in the area of object oriented software migration can be found in [22] and in [12].

Our approach differs from existing approaches in that we adopt an incremental and iterative object discovery process. The first step of our proposed process parses procedural legacy source code and creates an annotated AST, which is stored in an object-oriented database. Given the annotated ASTs, the global data types and formal parameters are then analyzed, and a set of candidate object models is built. Each object model includes a collection of (proposed) classes and associated methods. However, a method may be attached to more than one class. This leads to conflict-

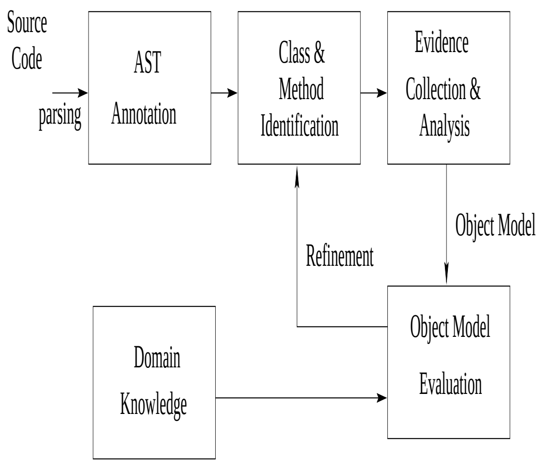


Figure 1: Overview of the object model recovery process.

ing models which are resolved by a followup refinement step. This step evaluates the candidate object models and consequently selects the most promising one. The selection and refinement is based on the data flow and interface complexity analysis. An overview of the process is shown in Fig.1.

The process is largely automated, and allows for domain knowledge to be considered during the object identification process.

3 Migration to Object Oriented Architecture

First, the source code is parsed and annotated ASTs are created. Annotations on the ASTs include link information, usage information (fetches, stores), and metrics. Second, the global data types and formal parameters are analyzed and collected together as a preliminary candidate class pool. The final selection of classes and methods is done with the help of evidence model. An evidence model is a set of heuristics based on the object oriented programming principles. This evidence model is driven by various features obtained from the source code (represented as AST annotations). The evidence model helps to establish an optimal object model with respect to well defined software engineering design principles, such as cohesion, coupling, depth of inheritance, and average method complexity. The source code features used by the evidence model include function return types, state changes, number of uses of a candidate class instances, number of function calls and so on. A number of heuristics exploiting inheritance, polymorphism, and overloading for the target system have been identified and incorporated into the re-engineering tool. At any point of this process, the user can assess and evaluate the target object model proposed by the developed tool and edit the evolving model using a UML compliant editor [1].

[1] We use Together/C++ White-board edition as our UML tool. Together is trademark of Object International Software Ltd.

3.1 Object Identification

The first step towards the migration of a procedural system to an object-oriented system is the selection of candidate object classes. This task can be automated to a large extend using a number of different software analysis techniques. However, no matter how sophisticated the analysis techniques are, user assistance and guidance is crucial in obtaining a viable and efficient object model. Significant domain information can be utilized by the user to guide the discovery process and direct it towards a better and more suitable object model. We have used the software analysis tool Refine[2]. to develop our prototype tools.

We use data type analysis as a principle strategy to identify objects in the procedural systems. We use two types of analysis. The first involves the analysis of global variables and their corresponding data types, while the second analyzes complex data types in formal parameter lists. The following subsections discuss these techniques in more detail.

3.2 Data Type Analysis

Analysis of global variables and their corresponding data types is based on the identification of variables that are globally visible within a module. A module is defined a set of consecutive program statements that deliver particular functionality and can be referred to by an aggregate name [9]. For our purposes, a module takes the form of a collection of functions. Any variable that is shared or is visible (i.e. it can be fetched and stored) by all the components of a module is considered as a global variable with respect to this module. Examples include `static` variables visible within a `C` source code file or `extern` variables shared between two or more `C` source code files. In order to perform this type of analysis, a set of modules have to be identified first. Clustering techniques and architectural recovery techniques presented in [6], [20], [10], [4] and [23] are used as a first step in order to obtain an initial decomposition of a large system in terms of module components. Clustering analysis yields modules that

[2] Refine is a Trademark of Reasoning Systems Inc.

are composed of functions which use or update a set of common global resources or reference common data types. Once a module has been identified then all aggregate data types associated with the module become primary object class candidates.

In addition to global data types, data type analysis is also focusing on aggregate type definitions that are present in formal parameter lists of the original system's functions. The pool of all data types that are identified by the global variable analysis and data type analysis constitutes the initial set of candidate classes. Once an initial set of classes has been identified method attachment follows. The process is incremental and iterative. In each iteration, the object model is refined and simplified. Possible refinements include class merging, formation of object hierarchies, and identification of possible polymorphic or overloaded functions. A number of heuristics have been added to the transformation tool to capture "good" design practices software engineers often use.

As an example of such a heuristic, consider the following declarations that may be obtained by different source files:

```
struct OBJECT {
  char name[MAXLEN];
  char idType[MAXLEN];
  char superClass[MAXLEN];
  char justification[MAXLEN];
  struct LIST *startList;
  struct ALIST ATTLIST[MAXATTRIBUTES];
  int numOfAtts;
  struct OBJECT *next;
};

typedef struct OBJECT OBJECT_TYPE;
```

In this case, the system identifies `OBJECT` and `OBJECT_TYPE` as one candidate class with the name `OBJECT`. The user may rename the candidate class through a menu driven user interface. The name change is recorded in a global table so that source code transformations for generating the new object oriented code will take these name mappings into account.

4 Method Identification

This part of the objectification process focuses on the discovery of functions or procedures from the original system which can be transformed to methods for the new object oriented system. Selecting methods and associating these methods to classes can be achieved by examining the formal parameters and usage patterns of aggregate data types in the functions of the original program. Basic types (i.e. `char`, `int`, `float` are ignored and only aggregate types (i.e `struct`, arrays) are considered. A special case applies to arrays of basic types which can become templates. A detailed description of this type of program transformation can be found in [7]. For the rest, if the complex and aggregate types are found in the parameter lists, the following simple rules are applied:

- For functions with no parameters, the return type and the type of global variables (in the scope of the particular function) that are modified/used become the initial candidate classes for which the particular function will become a method.

- For single parameter functions, the parameter type along with the return type and the global types modified become the candidate classes.

- For multiple parameter functions, all the parameter types along with the return type and the globally modified/used types are considered.

For a large number of functions, there is only one aggregate data type involved (in parameters, globally defined or as a return type) and these functions are immediately resolved as methods associated with the class derived from this aggregate data type. However, there are cases when a method is identified as a candidate method to more than one class. In this case, we say that the method is in a *conflict*. The following sections discuss the different types of evidence gathered to resolve methods that are in conflict and to provide a ranking mechanism for the different design choices that may occur.

4.1 Return Type Analysis

This type of analysis provides evidence based on the return type of a function. The motivation for using this criterion is that a return type usually indicates state change for a given data type. Since we are interested in complying with the concept of information hiding for the new system, we focus on data types that are in the formal parameters and are modified in the function body. These data types become the primary candidate classes to which the method that corresponds to a given function will be assigned to. However, poorly written code or code with side effects may not necessarily imply that a return type corresponds to a formal parameter that has been modified. For these cases, the state modification analysis which is described in the following section is considered.

4.2 State Modification Analysis

State modification analysis is based on the principle of information hiding and functional cohesion. The premise is that we would like to have methods that modify the state of their own class and minimize the side effects (state changes) of other classes. For example, consider the following statements in a function with its corresponding type declarations:

```
void InsertNode( RootPtr Root,
                 NodePtr aNode)
 {
   ......
   Root->node = aNode;
   ......
 }
```

This will provide an evidence of state change for the formal parameter type `RootPtr` and use of the type Node. State dependency tables can also be constructed and provide an overall picture of the data type dependencies for a legacy application [3]. For our system, state modification analysis also involves possible state changes due to function calls in the original procedural source code. Transitive closures of data type state modification via function calls and parameter passing by reference is also considered. For example, if function `avlInsert` calls `btInstert` and `btInsert` modifies a data type which is associated with `avlInsert`, then this data type is also added in the modified data types list for `avlInsert`.

State change information is recorded using entity-relationship tuples specified in RSF. [10] These tuples have the form `<entity> modifies <entity>` and can be loaded in a relational database for further analysis if required.

4.3 Usage analysis

This type of analysis focuses on the selection of data types used in the body of a function and can be exported via parameter passing by reference, return expressions, or global variables. For this analysis, all aggregate data types that are involved in expressions, castings, or in indirect component selections which are exported from a function (via parameter passing or global variables) are considered. This type of analysis is useful for providing coupling analysis (data dependencies) and provides an overall data flow view of the different object oriented designs that are possible.

4.4 Metrics Analysis

Metrics analysis provides an useful mechanism for assessing the impact of alternative design decisions on the overall quality of the target system. This assessment helps in resolving conflicts when a method could be attached to more than one class. For this work, we focus on two well known data flow design metrics, the *Information Flow* and the *Function Point* metrics.

Function Point

The function point metric is a design level metric [18] and is associated with the degree of functionality that is delivered by a given software component. An informal description of the metric is given below:

$$\begin{cases} p_1 * |GLOBALS(a\_constr)| + \\ p_2 * (|GLOBALS\_UPDATED(a\_constr)| + \\ |PARMS\_BY\_REF\_UPDATED \\ (a\_constr)|) + \\ p_3 * |READ\_STATS(a\_constr)| + \\ p_4 * FILES\_OPENED(a\_constr) \end{cases}$$

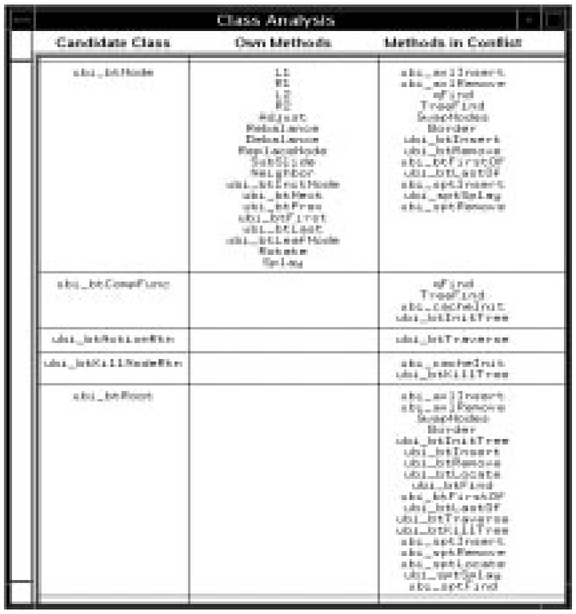
where $p_1, ..., p_4$ are integer coefficients. For our ranking purposes, this metric is an estimate of the functionality that can be delivered by a method when alternative designs are considered. In this context, we evaluate the metric for the method (that is the body of the original function) as if a choice has been made. That is, we pretend the method has been already attached to a class and the metric is computed on the premise that the function has been modified (i.e. the formal parameter list and its interface with the rest of the system has been changed). The metric is re-evaluated per alternative and the results are ranked. Overall, we are interested in minimizing the functionality delivered by a method because this leads to a modular design for the target system. That is, we are interested in designing methods that perform a specific task and are applied on as few data types as possible (ideally just one, in order to comply with functional cohesion) [9].

Information Flow

Information Flow is another useful data flow related metric and provides a measure of the interaction (fan-in, fan-out) of a software component with the rest of the system. Fan-in is defined as the number of data and control flows *terminating* at a module and fan-out is defined as the number of data and control flows *emanating* from a module. A more detailed description of this metric can be found in [18]. Similar to the *Function Point*, we are interested in computing the metric for each method and for each possible design choice (i.e. attachment of a given method to a class). The different alternatives are ranked and the one that minimizes *Information Flow* is considered as a primary candidate. The reason for minimizing the metric is that we would like to comply with the principles of information hiding and encapsulation. These principles suggest keeping data flows within the boundaries of a given unit (i.e. a class and the associated with it methods) and minimizing the unit's data interaction with the rest of the system.

4.5 Function Call Analysis

Function call analysis focuses on the examination of data types in the actual parameter lists of function calls that occur within a body of the function that is to be considered as a candidate method of a class. For example, if method M that corresponds to function F is in conflict and can be attached to different classes C_1, C_2, C_3 generated from data types T_1, T_2, T_3 respectively, function call analysis will examine the actual parameter lists of all function calls within the body of F. The data types that most often participate both in the formal parameter list of F and in actual parameter lists of calls within the body of F are considered primary candidate classes to attach method M. This type of analysis allows for collecting under a single class all methods that operate and alter the state of this class.

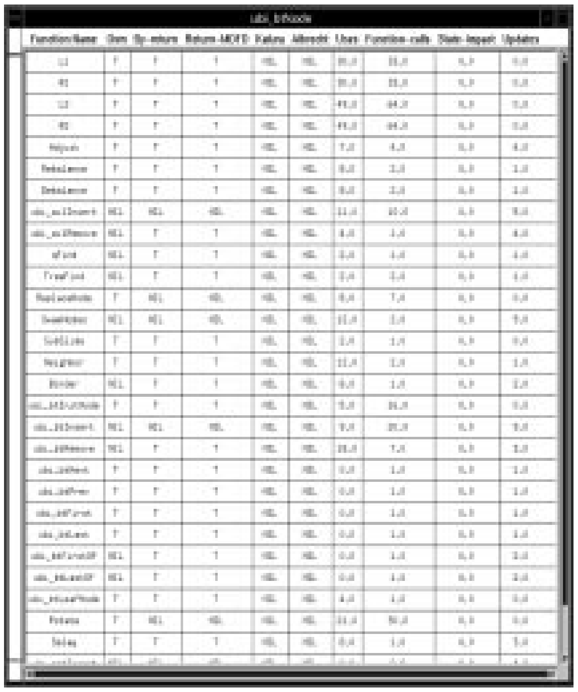


Candidate Class	Own Methods	Methods in Conflict
ubi_btNode	L1 R1 L2 R2 Adjust Rebalance Debalance ReplaceNode SubSlide Neighbor ubi_btInitNode ubi_btNext ubi_btPrev ubi_btFirst ubi_btLast ubi_btLeafNode Rotate Splay	ubi_avlInsert ubi_avlRemove qFind TreeFind SwapNodes Border ubi_btInsert ubi_btRemove ubi_btFirstOf ubi_btLastOf ubi_sptInsert ubi_sptSplay ubi_sptRemove
ubi_btCompFunc		qFind TreeFind ubi_cacheInit ubi_btInitTree
ubi_btActionRtn		ubi_btTraverse
ubi_btKillNodeRtn		ubi_cacheInit ubi_btKillTree
ubi_btRoot		ubi_avlInsert ubi_avlRemove SwapNodes Border ubi_btInitTree ubi_btInsert ubi_btRemove ubi_btLocate ubi_btFind ubi_btFirstOf ubi_btLastOf ubi_btTraverse ubi_btKillTree ubi_sptInsert ubi_sptRemove ubi_sptLocate ubi_sptSplay ubi_sptFind

Figure 2: Initial identification step provides candidate classes and methods. Before conflict resolution, a method could be a candidate to more than one class.

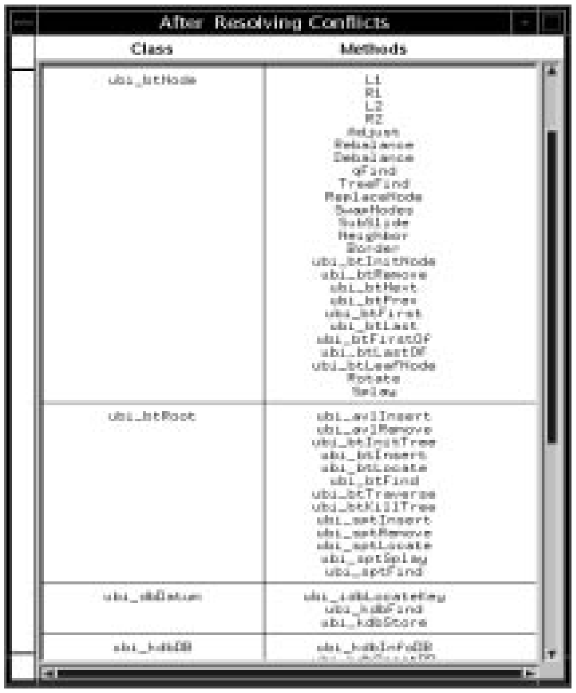
5 Method Attachment Resolution

The problem of discovering object oriented structures in procedural code may become a very complex task as the quality of the orig-


ubi_btNode

Function-Name	Dom	By-return	Return-MOFD	Kafura	Albrecht	Uses	Function-calls	State-impact	Updates
L1	T	T	T	NIL	NIL	10.0	11.0	0.0	0.0
R1	T	T	T	NIL	NIL	10.0	11.0	0.0	0.0
L2	T	T	T	NIL	NIL	41.0	64.0	0.0	0.0
R2	T	T	T	NIL	NIL	41.0	64.0	0.0	0.0
Adjust	T	T	T	NIL	NIL	7.0	4.0	0.0	4.0
Rebalance	T	T	T	NIL	NIL	8.0	2.0	0.0	1.0
Debalance	T	T	T	NIL	NIL	8.0	2.0	0.0	1.0
ubi_avlInsert	NIL	NIL	NIL	NIL	NIL	11.0	10.0	0.0	5.0
ubi_avlRemove	NIL	T	T	NIL	NIL	4.0	1.0	0.0	4.0
qFind	NIL	T	T	NIL	NIL	3.0	1.0	0.0	1.0
TreeFind	NIL	T	T	NIL	NIL	3.0	3.0	0.0	1.0
ReplaceNode	T	NIL	NIL	NIL	NIL	8.0	7.0	0.0	0.0
SwapNodes	NIL	NIL	NIL	NIL	NIL	11.0	3.0	0.0	5.0
SubSlide	T	T	T	NIL	NIL	3.0	1.0	0.0	0.0
Neighbor	T	T	T	NIL	NIL	11.0	1.0	0.0	1.0
Border	NIL	T	T	NIL	NIL	6.0	1.0	0.0	2.0
ubi_btInitNode	T	T	T	NIL	NIL	5.0	16.0	0.0	0.0
ubi_btInsert	NIL	NIL	NIL	NIL	NIL	9.0	20.0	0.0	5.0
ubi_btRemove	NIL	T	T	NIL	NIL	21.0	7.0	0.0	3.0
ubi_btNext	T	T	T	NIL	NIL	0.0	1.0	0.0	1.0
ubi_btPrev	T	T	T	NIL	NIL	0.0	1.0	0.0	1.0
ubi_btFirst	T	T	T	NIL	NIL	0.0	1.0	0.0	1.0
ubi_btLast	T	T	T	NIL	NIL	0.0	1.0	0.0	1.0
ubi_btFirstOf	NIL	T	T	NIL	NIL	0.0	1.0	0.0	2.0
ubi_btLastOf	NIL	T	T	NIL	NIL	0.0	1.0	0.0	2.0
ubi_btLeafNode	T	T	T	NIL	NIL	4.0	1.0	0.0	0.0
Rotate	T	NIL	NIL	NIL	NIL	11.0	50.0	0.0	0.0
Splay	T	T	T	NIL	NIL	8.0	1.0	0.0	5.0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 3: Evidence table for ubi_btNode class.


After Resolving Conflicts

Class	Methods
ubi_btNode	L1 R1 L2 R2 Adjust Rebalance Debalance qFind TreeFind ReplaceNode SwapNodes SubSlide Neighbor Border ubi_btInitNode ubi_btRemove ubi_btNext ubi_btPrev ubi_btFirst ubi_btLast ubi_btFirstOf ubi_btLastOf ubi_btLeafNode Rotate Splay
ubi_btRoot	ubi_avlInsert ubi_avlRemove ubi_btInitTree ubi_btInsert ubi_btLocate ubi_btFind ubi_btTraverse ubi_btKillTree ubi_sptInsert ubi_sptRemove ubi_sptLocate ubi_sptSplay ubi_sptFind
ubi_dbDatum	ubi_kdbLocateKey ubi_kdbFind ubi_kdbStore
ubi_kdbDB	ubi_kdbInfoDB

Figure 4: A resolution strategy based on global evidence provides a way of attaching methods to classes and eliminating conflicts.

inal system may be very poor. In Fig.2, part of the candidate object model for the AVL library is depicted. The model contains a number of conflicts. For example, method `ubi_avlInsert` is suggested as candidate method for `ubi_btNode` and `ubi_btRoot` classes. The conflict-resolution method is evidence-driven, and is based on the premise that a knowledgeable user can discover a better object-oriented design once he or she has a global view of the different design alternatives. Each design alternative can be ranked based on the impact it has on the fundamental source code features. We have selected eight source code features which are depicted in Fig.3 (columns 3-10).

The first column indicates the name of the potential method to be attached in `ubi_btNode`. The second column of the table indicates whether the method can be directly attached to a class or not. Here, a NIL value indicates that the method is a candidate for another class as well. The third column indicates whether the function returns the type for which the table is constructed and the fourth column indicates whether a parameter reference of this type is both modified and returned. The fifth and sixth columns indicate whether the Kafura and Albrecht metrics (Information Flow and Function Point metrics respectively) are minimized for a given method when this method is assigned to `ubi_btNode` class. The seventh column indicates the number of uses of `ubi_btNode` data type in the function body. The eighth column indicates the number of function calls within the function body that involve the `ubi_btNode` data type in their actual parameter list. The ninth column indicates whether there is a state change of the object for which this method will be attached to (i.e. the object associated with `this` keyword in `C`). Finally, the tenth column indicates the number of updates of type `ubi_btNode` which correspond to data that are local to the body of the function.

We construct one table per data type that has candidate methods in conflict. For example, Fig.3 provides evidence for `ubi_btNode` class. Each evidence can also be given a weight factor and the weighted sum to be used for ranking the different design alternatives. Our experimentation has revealed that *state impact* evidence along with the *return type* evidence are the significant factors for performing method attachment resolution. If these are not adequate to resolve method attachment, then *metrics* and *data type usage* could be the significant factors for ranking alternative designs. Fig.4 illustrates the results after applying the evidence driven conflict resolution techniques. In this case, the method `ubi_avlInsert` has been assigned correctly to class `ubi_btRoot`. In this example, the proposed model also allows for the polymorphic behavior of the insertion method in the case of library extensions (i.e. new insertion strategies applied to other types of binary trees such as binary heaps).

6 Source Code Generation

The main objective of this part of the process is to automatically generate code that can be compiled with minimal user modifications. Based on the class list and method resolution results, the header file (C++) is created first. Next, the header file is fed to the UML tool and relationships between classes are determined. The results from the method resolution process are examined and necessary changes are made. During the last step, the changes and relationships are incorporated in the source code to make the final design to be conformed with user's domain knowledge. The following two sections discuss this process in more details.

6.1 Generating Class Header Files

After class-method resolution, for each class from the class list, the corresponding class declaration C++ code is generated. The attributes of a class become private member variables for that class.

The functions in the original system are transformed into public methods of the class they are assigned to. If the class type appears in the function formal parameter list, and if that function is attached as a method to that class, the corresponding parameter is removed from the formal parameters. The functions with class type as a parameter are kept as non-static. The `static` keyword from C functions is removed. Conversely, the functions which return a type corresponding to their assigned class become static members. In order to access and manipulate member variables, additional methods (accessors and mutators) are created. As an example, consider the following function prototype,

```
ubi_btNodePtr ubi_RemoveTree
  (ubi_btRoot *RootPtr, ubi_btNode *Node)
```

that is transformed into a method as shown below.

```
ubi_btNode*
  ubi_btNode::
     ubi_btRemoveTree(ubi_btRoot *RootPtr)
```

6.2 Generating Class Implementation Files

In this step, class methods are implemented using the code from mapped C functions. For each class in the class list, the functions are traced from their AST and transformation routines are called to generate the corresponding method body in C++ syntax. As an example, consider the following procedure code fragment.

```
ubi_btNodePtr  ubi_avlRemove (
                 ubi_btRootPtr  RootPtr,
                 ubi_btNodePtr DeadNode)
{
  if ( ubi_btRemove(RootPtr,DeadNode))

    RootPtr->root =
           Debalance(RootPtr->root
           DeadNode->Link[01]
           DeadNode->gender);

    return (DeadNode);

}
```

If `ubi_avlRemove` is attached to `ubi_beNode` by the evidence model, then following piece of C++ code is generated.

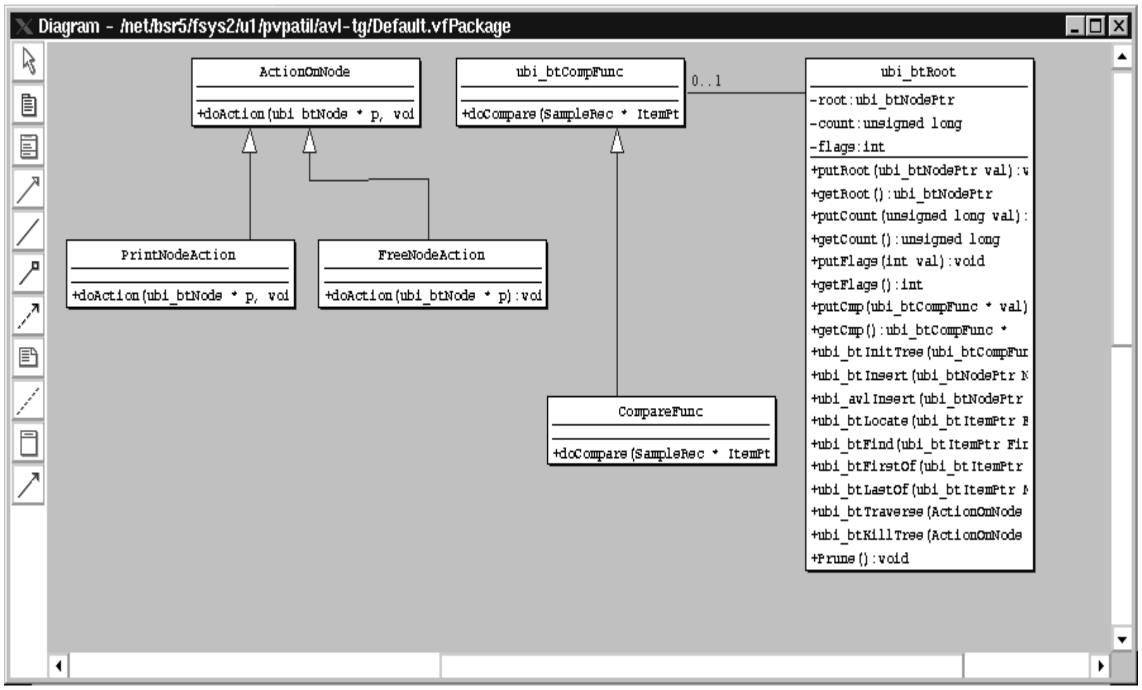


Diagram - /net/bsr5/fsys2/u1/pvpatil/avl-tg/Default.vfPackage
ActionOnNode
+doAction(ubi btNode * p, voi
ubi_btCompFunc
+doCompare(SampleRec * ItemPt
0..1
ubi_btRoot
-root:ubi_btNodePtr
-count:unsigned long
-flags:int
+putRoot(ubi_btNodePtr val):v
+getRoot():ubi_btNodePtr
+putCount(unsigned long val):
+getCount():unsigned long
+putFlags(int val):void
+getFlags():int
+putCmp(ubi_btCompFunc * val)
+getCmp():ubi_btCompFunc *
+ubi_btInitTree(ubi_btCompFur
+ubi_btInsert(ubi_btNodePtr N
+ubi_avlInsert(ubi_btNodePtr
+ubi_btLocate(ubi_btItemPtr F
+ubi_btFind(ubi_btItemPtr Fir
+ubi_btFirstOf(ubi_btItemPtr
+ubi_btLastOf(ubi_btItemPtr N
+ubi_btTraverse(ActionOnNode
+ubi_btKillTree(ActionOnNode
+Prune():void
PrintNodeAction
+doAction(ubi_btNode * p, voi
FreeNodeAction
+doAction(ubi_btNode * p):voi
CompareFunc
+doCompare(SampleRec * ItemPt

Figure 5: Refinement of class objects and their relationships with each other

```
ubi_btNodePtr
   ubi_btNode::ubi_avlRemove (
            ubi_btRootPtr  RootPtr )
{
  if ( ubi_btRemove(RootPtr))

  RootPtr->putroot(
     (RootPtr->getRoot())->
  Debalance(getLink(0x01),getgender()));

return  this;
}
```

In the method body, for accessing member variables of other class, accessors and mutators are used. These code generation utilities are incorporated in the basic tool of object identification. The code generation process traverses the ASTs for each class and generates syntactically correct C++ code for `.h` and `.cpp` files. This code includes automatically generated accessor and mutator methods for each class. In some cases, human intervention is required for obtaining a fully compilable code. However, this intervention involves minor changes and does not add significantly to the migration efforts.

7 Object Model Refinement

The resulting method-conflict resolution is presented to the user. Some methods can not be assigned automatically to just one class. Such methods have to be split into smaller methods and then can be assigned to appropriate classes. This process has to be done manually and user has to evaluate the code and make decisions about possible changes that he or she wants to implement. In order to assist the user in evaluation of generated code, the header files generated are exported to the object modeling tool(Together). Various relations between different classes are assigned to the class diagram. The main reason of using Together is that it supports UML and the changes in the object model are tracked back in the source code. In this step, the goal is to get an object model out of the generated class list and add new design elements from the domain-expert user if required.

In this work, we have identified a number of source code features that can help a developer make a design decision and obtain a better object oriented design from the original procedural code. Fig.5 shows a class diagram and of relations between classes. The user can edit these classes and their relations as these are presented to the user for further refinement.

In the following sections, we discuss a number of design decision rules that have been incorporated into the system to address issues related to inheritance, polymorphism, and method overloading.

Inheritance

Inheritance between object classes can be achieved by examining data structures in the original code. If two or more structures differ only with respect to few fields, these are candidate subclasses of a more general class. The more general class will contain the common fields and will inherit these fields with public inheritance to the subclasses. Certain functions copy one data structure to another data structure, which differs in many fields. The relation of these two data structures can be identified as inheritance. The structure with less fields becomes the base class and the other can be recognized as its derived class. Other evidence that supports inheritance is code cloning analysis, where two functions are identical with the only difference that they operate on different data types. Then these data types may become subclasses of a more general class (type) and the method can be attached and inherited from the more general class. Yet another heuristic that indicate possible inheritance is the presence of casting.

Overloading, Polymorphism

Overloaded methods can be also identified using code cloning analysis. If two or more functions are identified as clones with minor differences in their structure and the data types they use, then these functions can be overloaded on the data types they differ. The constraint is that these functions should return the same data type. Some groups of functions with similar names with the same prefix or suffix provide another clue to the name overloading. For example, execl(), execv(), execlp(), execvp() are functions to execute processes in various ways and all can be overloaded with the same function name. The functions with union type parameter become candidate overloaded methods. The reason is that these functions usually have different behavior according to the type of the union parameter they are applied upon. To simulate this behavior at the target system, these functions can be transformed into several overloaded methods with different parameter types that are obtained from the original `union struct` definition. Each overloaded method operates on the specific case of the original union structure. Similarly, polymorphic functions can be identified by examining function parameters that are pointers to functions. In this case, each possible function reference can become a class and their corresponding source code becomes a polymorphic method. As in example given below, consider the case of a tree traversal function that also takes a pointer to a function as a parameter that performs an operation on the node that is visited. Assuming that there is one tree traversal strategy, the tree traversal function may become a method attached to the tree class, and each action to be performed upon visiting a tree node may become a polymorphic method to a general `Action` class.

For example, we may have an `ActionOnNode` class with subclasses `PrintNodeAction`, `SwapNodeAction` and one polymorphic method called `DoAction()`. The following example illustrates this case which is a standard design pattern that is incorporated in the tool.

```
ubi_trBool ubi_btTraverse(
      ubi_btRootPtr   RootPtr,
      ubi_btActionRtn EachNode,
      void           *UserData )
```

The object recovery tool yields:

```
ubi_trBool ubi_btRoot:: ubi_btTraverse(
                   ActionOnNode * act,
                   void  *UserData)
```

where `ActionOnNode` is:

```
class ActionOnNode {
  public:
      virtual void doAction(
              ubi_btNode* p,
              void *UserData)= 0;
};
```

A class that implements the printing of the data items in a tree node can be defined as:

```
class PrintNodeAction:
             public ActionOnNode
{
  public:
     virtual void doAction(
         ubi_btNode* p, void *UserData)
    {
     p->PrintNode(UserData);
    };
};
```

8 Migration to Network-Centric Environment

Different object models arise from different requirements. C++ is a language for object-oriented systems and provides means for a component based infrastructure. On the other hand, emerging standards such as CORBA and IDL provide language-independent interface specification and component integration [21]. Once a legacy system has been re-engineered and an object model has been extracted, the system's original scope and usability can be greatly extended by allowing the re-engineered system to be in a form that supports distributed object orientation. In such a way, new functionality can be added to the re-engineered system in terms of plug-in components. In addition, existing parts of the re-engineered systems can be available from within other legacy applications, if need be. Wrapping the legacy components provides a low cost, low risk solution to achieve this target[5]. Each legacy component is encapsulated into the corresponding wrapper object. In the distributed computing environment, the wrappers realize the interfaces to translate the message passing between the calling and the called objects, redirect the innovation to the actual legacy class method. The wrapped components can be collected as the object repository, distributed on different servers.

In our approach, we take advantage of the CORBA and OMG IDL to accomplish the object wrapping task in terms of the following steps.

First, in order to provide the client program with a visible interface to other applications, each identified legacy class is defined as an IDL interface specification. Public methods are registered as the operations in the interface. Public data members are changed into the accessors and mutator operations in the interface. This IDL specification can be automatically generated by our tool and the object model.

Second, the CORBA IDL compiler translates the given IDL specification into a language specific (e.g. C++), client-side stub classes and server-side skeleton classes. Client stub classes and server skeleton classes are generated automatically from corresponding IDL interface specification. The client stub classes are proxies that allow a request invocation to be made via a normal local function but it represents the remote target object to the local applications. Server-side skeleton classes allow a request invocation received by the server to be dispatched to the appropriate servant. The operations registered in the interface become pure virtual functions in the skeleton class.

Third, wrapper classes are generated and implemented as CORBA objects, directly inheriting from the skeleton classes. This is a process that is also automatic, and is supported by our tool. The wrapper classes encapsulate the legacy object by reference, and incarnate the virtual functions by redirecting them to the encapsulated legacy class methods. The new functionality of the legacy object can be added in the wrapper class as long as the method name is registered in the interface.

For example, the `TreeNodeData` class, shown in Program1, is one of the classes identified by the analysis of the *GNU* tree libraries. This class is made visible to its remote clients by defining the interface `corba_TreeNodeData` in IDL, as shown in Program2. In Program3, `corba_TreeNodeData` interface is mapped to the skeleton class with the same identifier.The `wrapper_TreeNodeData` class inherits from `corba_TreeNodeData` and references the TreeNodeData class as the private data member. Examples on how encapsulation and wrapping can be achieved are shown below.

Program1: TreeNodeData class definition

```
class TreeNodeData {

      private:
         ubi_btNode * Node;
         char       Data[NAMESIZE];
         int        dataCount;

      public:
         TreeNodeData();
         void putData(char * val);
         char * getData();
         char  getData(int i);
         void putNode(ubi_btNode *val);
         ubi_btNode * getNode();
         void putDataCount(int aVal);
 };
```

Program2: corba_TreeNodeData interface definition

```
interface corba_TreeNodeData{
    void putData(in string val);
    string getDataString();
    char getData(in long i);
    // Other operations are eliminated
 };
```

Program3: wrapper_TreeNodeData class definition

```
class wrapper_TreeNodeData :
          public corba_TreeNodeData {
 private:
    CORBA::Boolean _rel_flag;
    TreeNodeData& _ref;
    char *_obj_name;

 public:
    wrapper_TreeNodeData(
      TreeNodeData& _t,
      const char *obj_name=(char*)NULL,
      CORBA::Boolean _r_f=0);

    wrapper_TreeNodeData(
      TreeNodeData& _t,
      const char *_serv_name,
      const CORBA::ReferenceData& _id,
      CORBA::Boolean _r_f=0);

    ~wrapper_TreeNodeData();

    CORBA::Boolean rel_flag();

    void rel_flag(CORBA::Boolean _r_f);

    void putDataCount(CORBA::Long aVal);

    // other function definitions
    ......
    // message transformation
    TreeNodeData* transIDLToObj (
           corba_TreeNodeData obj);
}
```

Since IDL does not support overloading and polymorphism, each method and data field within the interface should have an unique identifier. Otherwise it would be ambiguous when mapping to different languages. For example, C++ supports overloading, but C does not. To avoid changing the identified objects and to utilize the functionality offered by the overloaded and polymorphic methods, it is necessary to rename these methods by adding the prefix or suffix to the original name when they are registered in the interface. This "naming" technique will allow for unique naming conventions throughout the system, without violating code style standards. The wrapper classes are responsible for directing the renamed overloaded and polymorphic methods to the corresponding client code. When a client invokes a method through CORBA, it passes the CORBA data type parameters. The wrapper classes need the translation of the CORBA specific data types from the client calls to the data types used by legacy classes. Program4 illustrates the transformation from the CORBA specific type such as `corba_TreeNodeData_ptr` to the `TreeNodeData` used in the legacy function. In the same way, the wrapper classes convert the returned values from the legacy object to the CORBA specific data type. A wrapper class for the `TreeNodeData` class is illustrated in the example code below.

Program4: Message translation example

```
TreeNodeData* wrapper_TreeNodeData::
        transIDLToObj(
             corba_TreeNodeData obj)
{
   if (CORBA::is_nil(obj))
       return NULL;
```

```
    // Copy the data in the CORBA
    //object to the legacy object
    _ref.putData(obj->getDataString());
    _ref.putDataCount(obj->getDataCount());
    ubi_btNode *NodeImp = new ubi_btNode();
    wrapper_ubi_btNode
          NodeWrap(*NodeImp, _obj_name);
    _ref.putNode(NodeWrap.transIDLToObj
                (obj->getNode()));
    return &_ref;
};
```

9 Experiments

Experimentation for the proposed tool was carried out with three different C-based software systems. These included the expert system shell CLIPS, the GNU binary tree libraries (AVL, Binary Search Trees), and modules from a speech recognition system. The user interface and a set of APIs providing means for the user to customize and refine the object model has also been built in Refine / Intervista. A tool called Together/C++ is used for editing the object model.

The first experiment focused on computing the contribution of the individual source code features and providing an experimental view of the importance of each of different source code features during the object discovery process. The results were obtained by checking manually migrated source code, from three different systems against the code that has been generated by the system. The total size of the three system parts we experimented with was 12.5KLOC, consisting of 235 functions and 53 major data types. Manual migration was performed independently as part of another project.

The evaluation results are illustrated in Table.1. Based on the class list and method resolution, class declarations are created into a file which is passed on to the object visualization tool. The class list is then augmented with features such as an inheritance, generalization, dependency, along with relations between objects. The generated source code can be fine-tuned by the user and tested for a number of software quality features. In the context of maintainability, the comparison between quality-based metrics of the old and new system gives us an indication of the degree to which our chosen objectification heuristics are beneficial from a maintenance perspective. For our work, the quality of the new system is evaluated in terms of the metrics such as method complexity, weighted methods per class, depth of inheritance, coupling between objects and line of code per method.[1][14] Preliminary results suggest that the new system is indeed more maintainable than its forefather. The second experiment focused on the development of an object model for the three selected systems. One of the systems analyzed was a public library written in C for Sparse Arrays, AVL, Splay Trees, and Binary Search Trees [2]. The library also includes code for implementing simple and doubly linked lists and has a total size of 7.4KLOC. The original system was organized around C `structs` and a quite elaborate set of macros for implementing tree traversals and simulate polymorphism for insert, delete and tree balancing routines.

Feature	**Correct**	**False Negative**
State Change	36%	3%
Data Type Uses	26%	3%
Information Flow	17%	N/A
Return Types Modified	4%	4%
Return Types	1%	4%
Function Point	1%	N/A

Table 1: Impact of source code features on the method conflict resolution process.

The proposed system has been applied to the library, identifying on the first step 19 classes and 69 possible methods with a method conflict ratio of 49%. The second step was to apply the conflict resolution and class refinement strategy. Once source code for the target system has been generated then a UML editor allows the user to modify the object model, and let the tool re-generate the IDL and wrapper classes.

Obviously, remote invocation cannot have the equivalent performance as the local innovation. But the trade-off is the ability of moving parts of the new system to a faster server, or by making available the functionality of the original systems to other applications (open system).

Our experimental results suggest that the tool is scalable as it take approximately 650 seconds to generate a complete candidate object model for a 35KLOC system (CLIPS), running on an Ultra 1 Sparc station. Moreover, the migration can be done incrementally, without the need of obtaining a complete object model for the whole legacy system at once. On going experimentation involves the assessment of the re-usability and maintainability indices of the re-engineered system as these are compared to the original procedural system. Also, we work on the application of this technique to larger scale procedural systems.

10 Conclusions

We have presented techniques for re-engineering procedural legacy software systems into object-oriented, Network-Centric Environment. The migration of procedural code to object oriented architecture makes it more flexible to adopt for new requirements.

Once the legacy component has been encapsulated with a layer of CORBA wrapper class, it becomes very easy to enhance it with additional functionality, and to inter-operate with other completely independent applications. For example, client code can be Java applet and it can access the Java CORBA object server or C++ CORBA object server without knowing the details.

The techniques discussed in this paper have been evaluated with prototype tools that have been implemented and tested on medium size C systems.

Acknowledgments

The authors would like to thank Bill O'Farrell and Stephen Perelgut of IBM Center for Advanced Studies for their technical and management support without which this effort would not be possible.

About the authors

Prashant Patil is a graduate student at University of Waterloo and his research interests are reverse engineering and software migration. Ying Zou is a Ph.D candidate at the Electrical and Computer Engineering Department, University of Waterloo. Her research interests include distributed object technology, software re-engineering. Kostas Kontogiannis is an Assistant Professor at Electrical and Computer Engineering Department, University of Waterloo. His research interests include software re-engineering, software migration, software reuse and knowledge based software engineering. John Mylopoulos is a Professor at the University of Toronto, department of Computer Science. His interests include software requirements, conceptual modeling, software repositories and software re-engineering.

References

[1] Chidamber S. Kemerer C. A metrics suite for object oriented design. In *In Proceedings of IEEE TSE'94, vol.20*, pages 476–493, June 1994.

[2] Hertel C. http://www.interads.co.uk/ crh/ubiqx.

[3] Canfora G. et. al. Decomposing legacy programs:a first step towards migrating to client-server platforms. In *In Proceedings of IEEE IWPC'98*, pages 136–144, June 1998.

[4] Chase M. et.al. Analysis and presentation of recovered software architectures. In *In Proceedings of WCRE'96*, pages 153–162, November 1996.

[5] Cimitile A. et.al. Incremental migration strategies: Data flow analysis for wrapping. In *In Proceedings of WCRE'98*, pages 59–68, October 1998.

[6] Finnigan P. et.al. The software bookshelf. *IBM Systems Journal, vol.36*, 1997.

[7] Kontogiannis K. et.al. Code migration through transformations: An experience report. In *In Proceedings of IBM CASCON'98 Conference*, pages 1–13, Toronto ON, December 1998.

[8] Ogando R. et.al. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance:Research and Practice, Vol.6*, pages 261–283, 1994.

[9] Shari et.al. *Software Engineering, Theory and Practice.* Prentice Hall, 1998.

[10] Tilley S. et.al. Programmable reverse engineering. In *International Journal of Software Engineering and Knowledge Engineering*, pages 501–520, December 1994.

[11] Yeh A. et.al. Recovering abstract data types and object instances from conventional procedural language. *IEEE Software*, pages 227–236, 1995.

[12] Jacobson I. Lindstrom F. Re-engineering of old systems to an object-oriented architecture. In *In Proceedings of OOPSLA'91*, pages 340–350, 1991.

[13] Newcomb P. Kotik G. Reengineering procedural into object-oriented systems. In *In Proceedings of WCRE'95*, pages 237–249, 1995.

[14] Li W. Sallie H. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, pages 111–122, 1993.

[15] Sneed H. Encapsulating legacy software for use in client/server systems. In *In Proceedings of IEEE WCRE'96*, pages 104–119, November 1996.

[16] Sneed H. Object-oriented cobol recycling. In *In Proceedings of IEEE WCRE'96*, pages 169–178, November 1996.

[17] Haughton H. Lano K. Objects revisited. In *In Proceedings of IEEE Conf. on Software Engineering*, pages 152–161, October 1991.

[18] Adamov R. Literature review on software metrics. *Institut fur Informatik der Universitat Zurich*, 1987.

[19] Gall H. Klosch R. Finding objects in procedural programs:an alternative approach. In *In Proceedings of WCRE'95*, pages 208–216, 1995.

[20] Tzerpos V. Holt R. Software botryology: Automatic clustering of software systems. In *In Proceedings of the International Workshop on Large-Scale Software Composition*, Vienna, August 1998.

[21] Henning M. Vinoski S. *Advanced CORBA Programming with C++.* Addision-Wesley, 1999.

[22] Livadas P. Johnson T. A new approach to finding objects in programs. *Journal of Software Maintenance:Research and Practice, vol.6*, pages 249–260, 1994.

[23] Selby R. Basili V. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering, Vol.17, No.2*, pages 141–152, February 1991.