Evidence Driven Object Identification in Procedural Code

Kostas Kontogiannis Prashant Patil
University of Waterloo
Dept. of Electrical & Computer Engineering
Waterloo, ON. N2L 3G1
Canada

Abstract

Software evolution is an integrated part of software maintenance. It may take the form of porting a legacy system to a new hardware platform operating system, translating the system to a new language, or rearchitecting the system to take advantage of new programming paradigms. This paper presents techniques for the identification and recognition of object oriented structures in legacy systems that have been implemented using a procedural language. The paper examines methods for the selection of object classes and the recovery of the possible associations between the recovered classes.

1 Introduction

It is estimated that there are approximately 800 billion lines of source code worldwide. Most of this code corresponds to legacy systems that have been written decades ago using obsolete programming languages and paradigms. One of the most important issues in keeping these legacy systems operational is the inter-operability with other software applications which may run in remote locations, in a totally distributed Network-Centric environment. Recent advances in Network-Centric computing allow for distributed applications to be built base on the concept of object brokers. Standards such as CORBA, DCOM, ActiveX have evolved from research prototypes to fully commercial products. In addition, Java offers a unique platform for deploying software applications in a variety of hardware and software operating environments.

In order to bridge the gap between the legacy systems that have been built decades ago and the modern technologies provided by Network-Centric computing, we have to allow for the legacy systems to communicate with their operating environment in an and flexible efficient way. One way of achieve this objective is to use object wrappers. Object wrappers allow for legacy system components to be encapsulated and behave as objects in an distributed object oriented environment. In this context, the components of the legacy system that compose a unit must be identified. One way to achieve this is to partition the legacy system using clustering, data flow, or control flow analysis techniques [Holt98],[Tzerpos98], [Tilley94]. Another way is to restructure the legacy system in an object oriented way. Once the legacy system is restructured in an object oriented way, component selection, encapsulation and wrapping can be easily facilitated. This paper examines different techniques that can be applied in order to reveal object oriented structures in a large procedural systems. In particular, it presents techniques which are based on data type analysis, data flow, and software metrics. The approach is based on the premise that a system can be restructured in several possible ways and that different design alternatives can be examined and ranked based on a vector of quantitative and qualitative source code feature values. In such a way the user can assess the impact of each restructuring design decision, so that he or she can make the best possible choice among the different design alternatives. The paper discusses techniques to identify object classes using legacy source code, transform existing functions or procedures to methods and consequently attach these methods to object classes. Object classes are identified by examining data types that are used either for global communication between legacy system modules, or as interfaces between legacy system components (functions, procedures). Method identification is based on the examination of parameter passing, side effects, metrics, and global data flow analysis. Source code features and software metrics are used to provide a global view of the different alternative design and code restructuring decisions that

*This work was funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto) and the National Research Council of Canada.

may occur when a method can be attached to more than one class. Different design alternatives are then ranked based on the impact they have on the characteristics (cohesion, coupling, interface complexity) of target system.

2 Related Work

The migration of procedural legacy systems to object oriented systems have been discussed by a number of research groups in the software engineering community. Overall, in the area of object oriented migration the research efforts have focused on two main issues. The first issue is the identification of objects and abstract data types (ADT's). In [Yeh95] an interactive system that finds candidate Abstract Data Types and object instances in procedural systems is presented. The basis of the analysis is the program's Abstract Syntax Tree. A *field reference* graph provides information on the dependencies and uses of different structure (or record) fields by various functions or procedures of the source system. In [Gall95] a system that allows for the identification of objects in procedural systems using domain knowledge and user interaction is presented. A reverse generated object model extracted form the source code is compared to an independently developed object model of the underlying system. In [Haughton91] clustering techniques are used to identify abstract data types from existing procedures. The technique is based on attaching data types to existing procedures. A collection of a data type with its associated operations is then identified as an ADT. In [Ogando94] an object identification method based on the global data and data types, and on the use of formal parameters is proposed. In [Sneed96a] a system for the identification of objects in COBOL programs is presented. The approach uses slicing techniques to identify all of the elementary operations which change the state of a data type.

The second issue deals with the reenginering and the restructuring parts or the whole of an existing application in an object oriented way. This includes source code transformations, clustering, wrapping and, integration with other systems. In [Newcomb95] a reengineering tool that allows for the transformation of redundant, duplicated, and similar data and processes to classes and methods. In [Canfora98] a tool that allows for the decomposition of large legacy system is presented. The technique can be used to re-engineer a system into a client-server platform. In [Sneed96b] a system for reengineering 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 [Livadas94] and in [Jacobs91].

Our approach differs from other approaches in the point that we adopt an incremental and iterative object recovery process using an evidence model. The first step is to select candidate objects by analyzing global data types and function formal parameters. Once an initial set of candidate objects have been identified, a set of supporting evidence is gathered. This evidence includes state change information, return types, and data flow patterns. Once an initial object model has been created and an initial method attachment has been created, the process iterates so that new refined classes be identified. Source code features are gathered for each possible design alternative and an optimal design with respect to cohesion and coupling is presented to the user. The process is largely automated, and allows for domain knowledge to be considered during the object identification process.

3 Object Identification

The first step towards the migration of a procedural system to an object-oriented system is the selection of possible 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 on obtaining a viable and efficient object model. Significant domain information can be utilized by the user to guide the discovery process and to obtain a better and more suitable object model. In this paper we focus on the software analysis techniques that can be both largely automated and also allow for user interaction.

The object identification techniques used in this paper focus on two areas: a) the analysis of global variables and their data types, b) the analysis of complex data types in formal parameter lists. The following subsections discuss these techniques in more detail.

3.1 Global Data Type Analysis

Analysis of global variables and their corresponding data types is focusing on the identification of variables that are globally visible within a module. A module is defined as a consecutive set of program state-

ments that deliver particular functionality and can be referred to by an aggregate name [Shari98]. For our purposes a module takes the form of a file or a collection of files, or a collection of functions. Any variable that is shared or is visible (that is, 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 file, or `extern` variables shared between two or more `C` source files. In order to perform this type of analysis, a set of modules have to identified first. Clustering techniques, and architectural recovery techniques presented in [Finni97] and [Selby91] are used as a first step in order to obtain an initial decomposition of a large system in terms of module components. Once a module has been identified then all the global variables with respect to this module are primary candidates for analysis. For each variable its corresponding type is extracted from the Abstract Syntax Tree, and a candidate object class is generated.

3.2 Data Type Analysis

Data type analysis is focusing on type definitions that are accessible via libraries. Examples include `typedef C` constructs. Data types that are used in formal parameter lists become also primary class candidates. The union of data types that are identified by the global variable analysis and data type analysis forms the initial pool 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 is refined, object classes are merged, and the model is simplified. As an example 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;
```

The system identifies and `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 Attachment

Attaching methods to the candidate classes is an iterative and incremental process that focuses on the discovery of associations between classes (data types) and methods (functions). This analysis focuses on the examination of the formal parameters 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 become templates. A detailed description of this type of program transformation can be found in [Konto98]. For the complex and aggregate types 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 this method.
- 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 a method is in conflict. The following sections discuss in more detail the different types of evidence gathered to resolve methods that are in conflict and provide a ranking mechanism and different design choices that may occur.

4.1 Return Type Analysis

This type of analysis provides evidence based on the return type of the 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 on complying with the concept of information hiding for the new system, we focus on data types that correspond to formal parameters modified. These data types become the the primary candidate class for hosting the method that corresponds to a function. 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)
 {
   Node *newNode = new Node(10);
 .......

 Root->node = newNode;
```

will provide evidence of state change for the formal parameter type `RootPtr` and use of the type Node. This will be supporting evidence that the given function be assigned as method to the class that corresponds to the modified data type `RootPtr`. State dependency tables can also be constructed and provide an overall picture of the data type dependencies for a legacy application [Canfora98]. For our system, state modification analysis also involves possible state changes due to function calls in the original procedural source code. Transitive closures of state modification via function calls and parameter passing by reference is also considered. For example if function `avlInsert` calls `btInstert` and `btInsert` modifies through parameter passing by reference a variable whose data type 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 [Tilley94]. These tuples have the form `<entity> modifies <entity>` and can be loaded in a relational database for further analysis if required.

4.3 Use analysis

This analysis focuses on the selection of data types used in the body of a function. For this analysis all the aggregate data that are involved in expressions, casting, or in indirect component selections and can be exported from a function (via parameter passing or global variables) are considered. This type of analysis is useful on providing an overall 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 another useful mechanism for ranking possible alternative designs and help resolve situations when a method could be attached in different classes. For this work we focus on data flow metrics and in particular the *Information Flow* and the *Function Point* metric.

Function Point

The function point metric is a design level metric [Adamov87], 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. The metric is re-evaluated per alternative and the results are ranked. Overall, we are interested on minimizing the functionality delivered by a method because this leads to a modular design for the target system. That is, methods that perform a specific task are applied on as few as possible data types (ideally just one, in order to comply with functional cohesion) [Shari98].

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 flows *emanating* from a module. A more detailed description of this metric can be found in [Adamov87]. Similarly to the Function Point, we are interested on 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, that is to keep data flows within the boundaries of a given module (i.e. a class and the associated with it methods). The motivation is to minimize a module'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 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 formal parameter lists of all function calls within the body of F. The data types that most often participate both on the formal parameter list of F and on 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 the class.

5 Inheritance, Polymorphism, Basic Patterns

Class identification and method attachment are tasks that can be automated to a large extent. However, obtaining a good design is a task that requires user assistance. In this work we have identified a number of source code features that can help a developer obtain a better object oriented design from the original procedural code.

Inheritance

Inheritance between object classes can be achieved by examining data structures in the original code. If two or more structures differ on a few fields, these are candidate types to become subclasses to a more general class. The more general class will contain the common fields and will inherit these fields with public inheritance to the subclasses. 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.

For example the candidate classes `OBJECT`, `CLASS`, `SUBCLASS`, `MEMBER`, may become one class or be organized in an hierarchy with `OBJECT` being the superclass and the other three inherit from it.

```
struct OBJECT
 { char name[MAXLEN];
   char idType[MAXLEN];
   struct LIST *startList;
   int numOfAtts;
   struct OBJECT *next;
  };
typedef struct OBJECT CLASS;
typedef struct OBJECT SUBCLASS;
typedef struct OBJECT MEMBER;
```

The different design alternatives are assessed according to the method attachment choices made for these candidate classes. For example if there are methods that operate only in some of the candidate classes, and some method found to be common, an object hierarchy as suggested above will be formed.

Polymorphism, Overloading

Overloaded methods can be identified using code cloning analysis. If two or more functions are identified as clones with minor differences in their structure and on the data types they use, then these functions can can be overloaded on the data types they differ. The constraint is that these functions should return the same data type. In Fig.1, the results of applying the clone detection analysis is presented. These results help identify potential overloaded methods.

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 an example consider the case of a tree traversal function that also takes as a parameter a pointer to a function that performs an operation on the node that is visited.

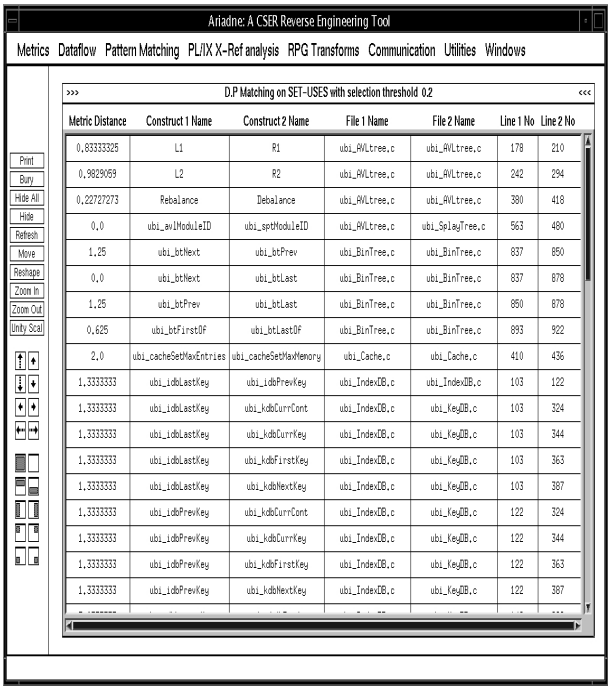

Figure 1: Code cloning analysis. numeric values provide a measure of difference.

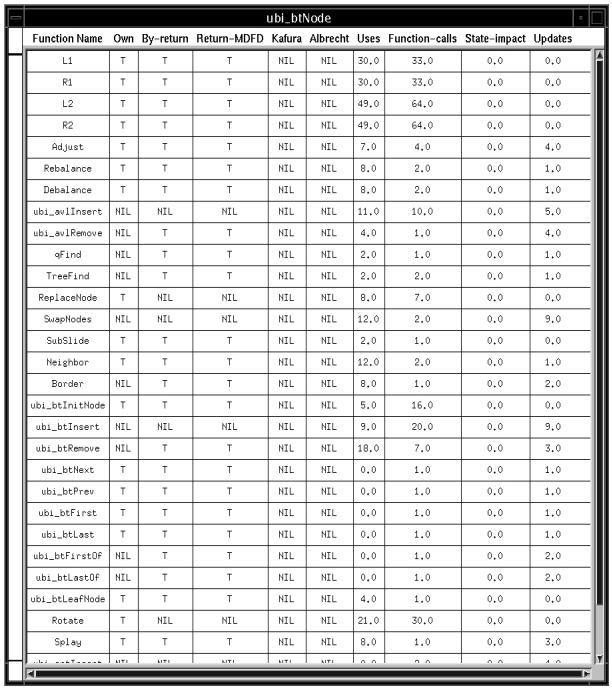

Figure 2: Evidence table for ubi_btNode class.

The tree traversal function may become a method for the tree class, and each action may become a polymorphic method to a corresponding 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 corresponding migration 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 );
    };
};
```

6 Method Attachment Resolution

The problem of discovering object oriented structures in procedural code may become a very complex task as the quality of the original system may be very poor. For example the use of global variables, side effects and, code cloning to mention a few, can result on misleading source code features and therefore to erroneous object oriented structures will be identified. The proposed method is evidence driven and is based on the premise that a 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 fundamental source code features.

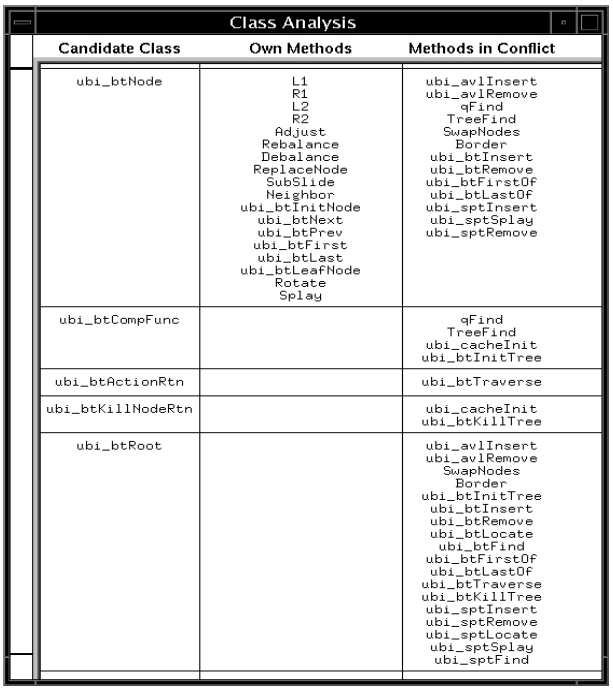

Figure 3: Initial identification step provides candidate classes and methods. Before conflict resolution, a method could be a candidate to more than one class.

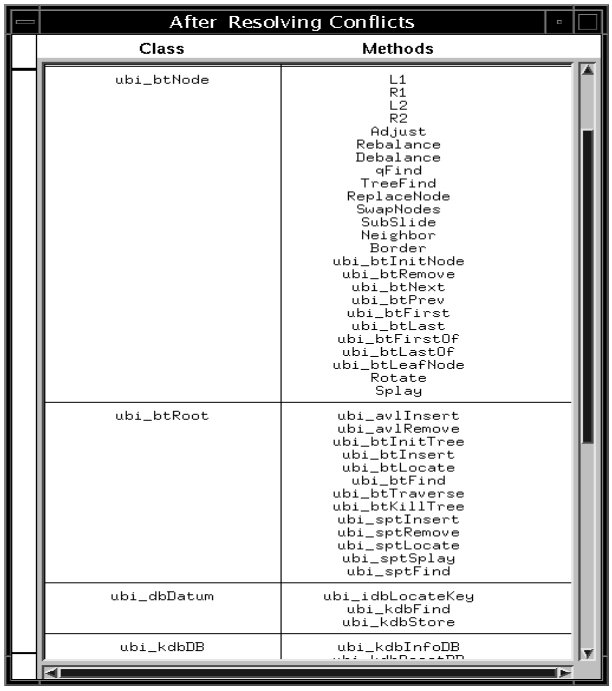

Figure 4: A resolution strategy based on global evidence provides a way of attaching methods to classes and eliminating conflicts.

The resolution strategy is based on a global evidence table per candidate class such as the one presented in Fig.2. We construct one table per data type that has candidate methods in conflict. For example Fig.2 provides evidence for `ubi_btNode` class. The first column indicates the name of the potential method to be attached in `ubi_btNode` The second column of the table indicates if the method can be directly attached to a class or not. A NIL value indicates that the method is a candidate for another class as well. The third column indicates if the function returns the type for which the table is constructed, and the fourth column indicates if a parameter reference of this type is both modified and returned. The fifth and sixth columns indicate if Albrecht and Kafura metric (Function Point and Information Flow metrics respectively) are minimized when a given method is assigned to `ubi_btNode` class. The seventh column indicates the number of uses of `ubi_btNode` data type in the body of the method. The eighth column indicates the number of function calls within the body of the method that involve the `ubi_btNode` data type in their actual parameter list. The ninth column indicates whether or not 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.

Each evidence can also be given a weight factor and the weighted sum is used to rank the alternatives. Our experimentation has revealed that *state change* 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.

7 Experiments

For our experiments we used a number of different C-based systems. These included the expert system shell CLIPS, binary tree libraries (AVL, Binary Search Trees), modules from a proprietary speech recognition software, and two UNIX shells (bash, tcsh). Manual inspection classified the results in two categories *1)* Correct (a method has been correctly identified and attached to the right class), *2)* False Negative (a method has been missed, that is it has not been identified and attached to a class that it should belong to). False Positives can not be easily identified as there

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.

may be many alternative designs that are correct and acceptable.

The first experiment focused on computing the contribution of the individual source code features and provide an experimental view of the importance of each source code feature for 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 sample systems was 8.5KLOC, consisting of 235 functions and 53 major data types. The manual migration was performed independently as part of another project. The evaluation results are illustrated in Table.1 [1].

The results indicate that the state change feature is the one that dominates the method resolution process. It has the highest impact among the other features, and at the same time keeps the percentage of False negatives low. The number of data type uses also is an important factor on method conflict resolution. Finally, the Information Flow metric was identified as the third best source code feature to be used when developing an object model. More experiments are under way in order to cover more possible migration cases that may occur in a large system. However, we believe that these results provide an initial road map for software practitioners who are involved in migrating procedural code to object oriented environments.

The second experiment was to examine the impact of the migration process on a number of sample systems written in C and infer a new Object Oriented architecture. One of the systems analyzed was a public library written in C for Sparse Arrays, AVL, Splay Trees and, Binary Search Trees [Hertel98]. The library also includes code for implementing simple and doubly linked lists. The original system was organized around

[1] The "N/A" ientry in the table indicates that there were not enough cases in test data to provide a statistically significant result.

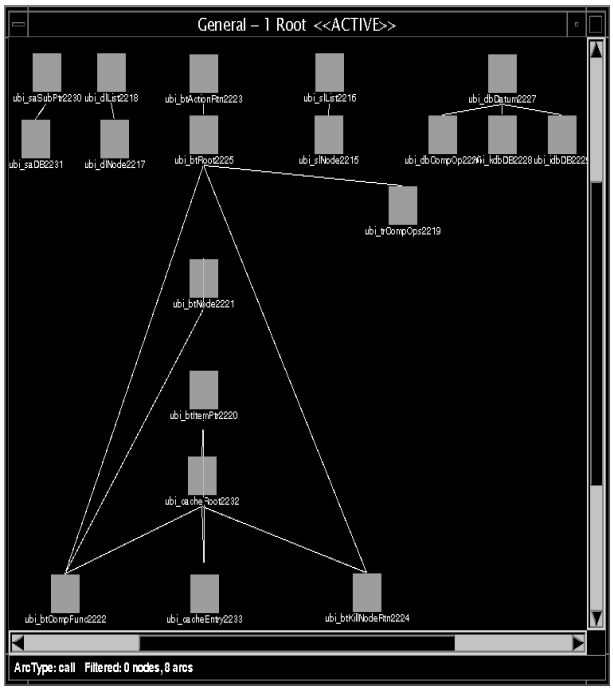

Figure 5: Tree library object model (uses view)

C `structs` and a quite elaborate set of macros for implementing tree traversals and simulate Polymorphism for insert, delete and tree balancing routines.

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 strategy and visualize the library architecture from the object oriented point of view. The object model for the library is depicted in Fig.5 and in Fig.6. In Fig.5 nodes represent classes and arcs represent method invocations. That is an arc from node `A` to node `B` implies that there class `A` is a client of `class B`. Similarly, in Fig.5 data flow information is depicted. In particular, nodes represent classes and arcs represent data uses. That is an arc from node `A` to node `B` implies that there is an instance of class `B` used in class `A`. Fig.6 and Fig.5 indicate the existence of 5 subsystems (clusters on the top). The leftmost (upper left corner) is the subsystem that now corresponds to all classes and methods for the Single Linked List library (`ub_slList`, and `ubi_slNode`. The second subsystem indicates the Doubly Linked List Library. The third subsystem corresponds to Sparse Arrays. The fourth subsystem corresponds to Binary Trees (including AVL, Binary Search Trees, and Sparse Trees). Finally the fifth subsystem corresponds with the utilities and classes for

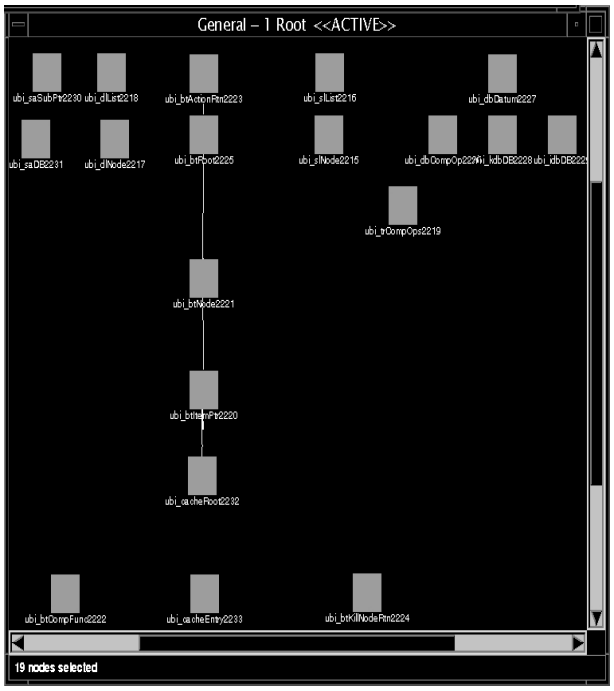

Figure 6: Tree library object model (calls view)

attaching data items to the nodes of the data types implemented by the library (e.g. data values attached to the nodes of the AVL trees).

8 Conclusion

In this paper a method for the identification of object models in procedural legacy systems is presented. The method is based on an iterative and incremental process. The process starts by identifying an initial set of candidate classes and a set of candidate methods, by analyzing data types and formal parameters. The process iterates by refining the initial object model. The refinement is based on selecting a set of source code features that need be optimized. These code features include the number of object state changes, data flow dependencies, interface complexity (fan-in, fan-out), uses and updates of data types. The system allows for major subsystems to be identified, and facilitate the encapsulation of the identified subsystems in wrapper classes in the case the original system is migrated to a Network-Centric environment.

References

[Adamov87] Adamov, R. "Literature review on software metrics", *Institut fur Informatik der Universitat Zurich*, Switzerland, 1987.

[Canfora98] Canfora, G., Cimitile, A., DeLuccia, A., DiLicca, A., "Decomposing Legacy Programs:A First Step Towards Migrating to Client-Server Platforms", *In Proceedings of IEEE IWPC'98*, June 1998, pp.136-144.

[Finni97] Finnigan , P. et.al "The Software Bookshelf", *IBM Systems Journal*, vol.36, No.4, 1997.

[Gall95] Gall, H., Klosch, R., "Finding Objects in Procedural Programs:An Alternative Approach", *In Proceedings of WCRE'95*, pp.208-216.

[Haughton91] Haughton, H., Lano, K., "Objects revisited", *In Proceedings of IEEE Conf. on Software Engineering*, October 1991, pp.152-161.

[Hertel98] Hertel, C., http://www.interads.co.uk/ crh/ubiqx/

[Holt98] Holt, R., "Structural Manipulations of Software Architecture using Tarski Relational Algebra", *In Proceedings of WCRE: Working Conference on Reverse Engineering*, Honolulu, Oct 1998.

[Jacobs91] Jacobson, I., Lindstrom, F., "Re-engineering of old systems to an object-oriented architecture", *In Proceedings of OOPSLA'91*, pp.340-350.

[Konto98] Kontogiannis K., martin, J., Wong, K., Gregory, R., Muller, H., Mylopoulos, J., "Code Migration Through Transformations: An Experience Report", *In Proceedings of IBM CASCON'98 Conference*, December 1998, Toronto ON..

[Livadas94] Livadas. P., Johnson, T., "A New Approach to Finding Objects in Programs", *Journal of Software Maintenance:Research and Practice*, Vol.6, 1994, pp.249-260.

[Newcomb95] Newcomb P., Kotik, G., Reengineering Procedural into Object-Oriented Systems", *In Proceedings of WCRE'95*, pp.237-249.

[Ogando94] Ogando, R., Yeu, S., Liu, S., Wilde, N., "An Object Finder for Program Structure Understanding in Software Maintenance", *Journal of Software Maintenance:Research and Practice*, Vol.6, 1994, pp.261-283.

[Selby91] Selby R., Basili, V., "Analyzing Error-Prone System Structure", *IEEE Transactions on Software Engineering*, Vol.17, No.2, February 1991, pp. 141-152.

[Shari98] Shari, Lawrence, Pleeger, "Software Engineering, Theory and Practice", Prentice Hall, 1998.

[Sneed96a] Sneed, H., "Encapsulating Legacy Software for Use in Client/Server Systems", *In Proceedings of IEEE WCRE'96*, November 1996, pp.104-119.

[Sneed96b] Sneed, H., "Object-Oriented COBOL Recycling", *In Proceedings of IEEE WCRE'96*, November 1996, pp.169-178.

[Tilley94] Tilley S., Wong, K., Storey, M., Muller, H., "Programmable Reverse Engineering" *International Journal of Software Engineering and Knowledge Engineering*, pp. 501-520, December 1994.

[Tzerpos98] Tzerpos V., Holt, R., Software Botryology: Automatic Clustering of Software Systems, *In Proceedings of the International Workshop on Large-Scale Software Composition*, Vienna, August 1998.

[Yeh95] Yeh, A., Harris, D., Reubenstein, H., "Recovering Abstract data Types and Object Instances from Conventional Procedural language", *IEEE Software*, 1995, pp.227-236.