A Framework for Migrating Procedural Code to Object-Oriented Platforms

Ying Zou, Kostas Kontogiannis
Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada
{yzou, kostas}@swen.uwaterloo.ca

Abstract

With the rapid growth of the Internet and pervasive computing activities, the migration of back-end legacy systems to network centric environments has become a focal point for researchers and practitioners alike. To leverage back-end legacy services into Web-enabled environments, this paper proposes an incremental and iterative migration framework where legacy procedural source code is reengineered into an object-oriented platform. The reengineering framework allows for the representation of the legacy source in he form of XML based Annotated Abstract Syntax Trees. Consequently, the extraction of an object-oriented model from the original source code is based on the analysis of source code features in the original system that can be used to identify classes, associations, aggregations, and polymorphic patterns in the new target system.

1. Introduction

With the widespread use of the Internet and pervasive computing technologies, distributed object technologies have been widely adopted to construct network-centric architectures, using Web Services, CORBA, and DCOM. This use has triggered a plethora of research with the main objective to leverage the business value of legacy software systems into Web-enabled environments. Object oriented technologies play an important role in the reengineering, integration and deployment of back end legacy services into Web enabled platforms.

To facilitate the reengineering of legacy services in Web-enabled environments, our approach proposes a framework where components of a procedural egacy system can be migrated to an object-oriented platform. The proposed framework is based on a methodology that allows for the source code of a legacy system to be represented at a higher level of abstraction using an XML domain model, and second on an analysis and transformation technique that aims at extracting an object model from the procedural system. The target migrant system is intended to be more maintainable than the original system and to possess an open programmatic interface (API) that can be used for integration and deployment to a Web enabled environment. Such object components that deliver back-end services can also become building blocks of reference software architectures that are related to Web and pervasive computing applications.

The remainder of the paper is organized as follows. Section 2 reviews related works in literature. Section 3 gives an overview on the framework of object oriented model discovery process. Section 4 discusses the approach that represents C source code in XML. Section 5 proposes a catalog of evidences that qualify features in procedural code towards object-oriented model. Section 6 provides experimental results for the object-oriented model discovery. Finally, section 7 provides the conclusion of the paper and some pointers for the future research.

2. Related work

2.1. Source code representation

There is a growing stream of activities related to XML representation of source code. In [8], a system for annotating C++ and Java is presented. Specifically, Java and C++ grammars are mapped to corresponding DTDs using a domain model. Consequently, semantic actions have been added to custom made parsers in order to annotate the input stream (source code) with XML tags that are compliant to a domain model DTD. In this approach, common structures between object oriented languages are abstracted in a more generic DTD that aims to model object oriented language constructs.

)-7695-1083-3/01 $17.00 © 2001 IEEE

In [7], the InterMediate Language (IML) is proposed to model and analyze source code. IML allows for sophisticated data-flow and control-flow analyses to be built. Extensions to IML have been discussed in [13], where the Resource Graph (RG) is proposed to abstract global information, such as call, type, and usage relations for architectural design recovery.

In [6], he Graph Exchange Language (GXL) is proposed as a data exchange format among software analysis tools. GXL is designed for the representation of typed graphs.

Other program representation schemes, such as ASG, Program Dependence Graph, Rigid Standard Format (RSF), Tuple-Attribute Language (TA) and AsFix, represent the source code at different abstraction level, and are used in different program analysis tools. A summary of these techniques and a high-level exchange schema between them is discussed in [3].

2.2. Objectification

In the relevant literature, several methods for identifying an object model from a legacy system have been defined. Overall, these research efforts focus on the identification of objects and abstract data types (ADTs). In [4], the identification of object model from RPG programs is presented. Objects are centered around persistent data stores, while related chunks of code in the legacy system become candidate methods. In [9, 12], an object model is discovered directly from procedural code written in C. Candidate objects are selected by analyzing global data types and function formal parameters. An evidence model helps to attach the methods to a candidate class and choose an appropriate object model. This evidence model is consisted of state change information, return types, and data flow patterns.

Another objectification method is presented in [18]. The method is based on documentation and informal information, such as user manuals, requirement and design specifications, and naming conventions. However, for legacy systems, the external information is not always available, and this technique may not always be applicable. The technique may also be used to analyze source code informal information such as comments and identifier names and from other non-linguistic aspects of OO code.

Our approach extends the research presented in [9, 12]. Specifically, in addition to identifying objects based on abstract data types, we aim to provide an iterative and incremental process that allows for alternative object models to be extracted form the procedural source code and be evaluated conformance to specific software engineering principles namely cohesion and coupling. Moreover, we aim to discover an object model that incorporates the object-oriented characteristics of a well-designed system, namely the characteristics of class inheritance, polymorphism and overloading.

3. Portable source code representation

3.1. Abstract syntax tree

At the lowest level of abstraction, Abstract Syntax Trees (ASTs) have been successfully used by the data flow analysis community in order to analyze and transform source code entities. These trees contain information about the source program [6] in the form of nodes and edges. Such tree-like structures represent the source program in a top-down matter. For example, C applications are represented at the top level as `applications`, `modules`, and `files`, while at lowest levels as `functions`, `declarations`, `macros`, `expressions`, and `identifiers` to name a few [1]. The internal nodes of the AST represent the non-terminal phrases such as `statements`, `operations`, `functions`, and the leaf nodes represent the terminal symbols, such as `identifiers`, and `empty declarators`. An edge denotes tree attributes, which are represented as mappings between AST nodes. AST nodes correspond to programming language constructs such as `If-Statements`, `For-Statements`, and `While-Statements`.

However, there are several issues that need be considered for the successful application of ASTs with the objective of developing flexible software analysis tools. Namely, these issues include:

1. Decoupling the AST representation from specific parsing environments by utilizing domain models for programming languages,
2. Allowing for AST representations to be extended in order to allow tools to be built in an incremental way,
3. Using a flexible, open and standard API so that such analysis tools can be integrated into collaborative software maintenance environments.

These minimal requirements could ensure that software re-engineering tools can be quickly built, easily maintained, and provide multiple views of the system being analyzed.

3.2. Advantages of representing AST in XML

XML provides a unified framework for annotating structured data. It is extensible, and can define meaningful tags that link syntax with the semantics of the entities for a given domain. Similar to AST, an XML document can be thought of as a tree structure with nodes and edges connected by a hierarchy relationship. Such a tree is called a Document Object Model (DOM) tree.

Compared to the custom-made software extractors, the

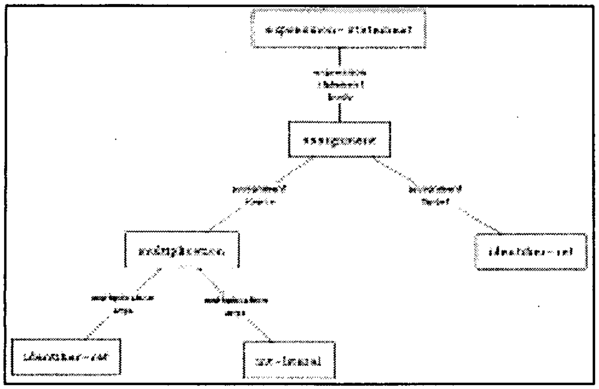


Figure 1. AST Structure for Expression Statement in C

DOM provides a standard API for programmatic access to XML documents, and manipulates the structural data. In such a way, tool developers can consistently interact with, and analyze XML-based documents. In addition, the DOM APIs are widely supported and can be bounded with different programming languages, such as Visual Basic, C++, Java, and JavaScript.

3.3. Mapping ASTs into XML

There are two main approaches to extract the Abstract Syntax Tree of a source code fragment and encode it in XML. The bottom-up approach utilizes the concept of a domain model definition that denotes the syntactic structures of a programming language such as Pascal, Fortran, C, C++ and Java. Tools that utilize this approach include Refine for C, Datrix for C++/C/Java, JavaCC for Java [8], and IBM VisualAge for C++ and Java [8].

The second approach, referred to as the top-down approach, examines the grammar of the specific programming language, and defines a standard logical structure for an annotated Abstract Syntax Tree. By following the rules, different parsers can extract the necessary information from the source code and encode it in a uniform and application-independent format. Using a domain model definition extracted from the specification of a given programming language (i.e. ANSI C), we employ a two step hybrid approach to define the logical structure of the entities of an Abstract Syntax Tree in terms of a Document Type Definition (DTD) document.

For our work, we define the structure of the Abstract Syntax Tree for a specific language by developing a domain model for this language. By recursively traversing the hierarchy of the domain model entities, we are able to map the given domain model to a Document Type Definition (DTD). Specifically, the tree hierarchical structure of the domain model is mapped to XML elements and attributes. Figure 1 illustrates an Abstract Syntax Tree that conforms to a given domain model and represents the C assignment expression, "shuffle_level = num_decks * 26". The non-terminal nodes are expression-statement, assignment, and multiplication. The leaf nodes represent terminal tokens, such as identifier-ref (identifier reference), int-literal (integer). Similarly the edges represent language construct attributes as mappings between AST nodes. For example, the edge, named assignment-source, is considered as an attribute of the assignment construct that contains as a value a node of type multiplication. Each node and edge in the AST is mapped to an XML element tag as illustrated in Figure 2. The attribute values of an AST node are mapped to the corresponding attribute values of the XML elements.

```
<EXPRESSION-STATEMENT >
  <EXPRESSION-STATEMENT-BODY>
   <ASSIGNMENT surface-syntax="shuffle_level = num_decks * 26">
    <ASSIGNMENT-TARGET surface-syntax="shuffle_level">
     <IDENTIFIER-REF id-name="shuffle_level"/>
    </ASSIGNMENT-TARGET>
    <ASSIGNMENT-SOURCE surface-syntax="num_decks * 26">
      <MULTIPLICATION surface-syntax="num_decks * 26">
       <MULTIPLICATION-ARGS>
         <IDENTIFIER-REF id-name="num_decks"/>
         <INT-LITERAL int-long="NIL" int-radix="10"
                           int-unsigned="NIL" value="26"/>
       </MULTIPLICATION-ARGS>
      </MULTIPLICATION>
    </ASSIGNMENT-SOURCE>
   </ASSIGNMENT>
  </EXPRESSION-STATEMENT-BODY>
</EXPRESSION-STATEMENT>
```

Figure 2. XML Element for Expression Statement in C

```
<?xml version="1.0"?>
<!element EXPRESSION-STATEMENT
                    (EXPRESSION-STATEMNT-BODY)>
<!element EXPRESSION-STATEMENT-BODY(EXPRESSION)>
<!element EXPRESSION(ASSIGNMENT |MULTIPLICATION |...)>
<!element ASSIGNMENT (ASSIGNMENT-TARGET,
                              ASSIGNMENT-SOURCE)>
<!element ASSIGNMENT-TRAGET
                    (EXPRESSION|IDENTIFIER-REF|... )>
<!element ASSIGNMENT-SOURCE
                    (EXPRESSION|IDENTIFIER-REF|...)>
<!element MULTIPLICATION(MULTIPLICATION-ARGS)*>
<!element MULTIPLICATION-ARGS
                    (IDENTIFIER-REF*|INT-LITERAL*|.....)>
<!attlist ASSIGNMENT surface-syntax  CDATA #REQURIED>
<!attlist ASSIGNMENT-TARGET surface-syntax CDATA #REQURIED>
<!attlist ASSIGNMENT-SOURCE  surface-syntax CDATA #REQURIED>
<!attlist MULTIPLICATION surface-syntax CDATA #REQURIED>
<!attlist IDENTIFIER-REF id-name CDATA #REQURIED>
<!attlist INT-LITERAL int-long CDATA #REQURIED
                      int-radix CDATA #REQURIED
                      int-unsigned CDATAD #REQURIED
                      value NUMBER  #REQURIED>
```

Figure 3. DTD for Expression Statement in C

Consequently, we focus on the enhancement and generalization of the domain model and common schema for the programming language being modeled. The domain model for a given language and its corresponding DTD can be enhanced with information such as unique identifier numbers, linkage, and analysis information. Similarly, domain model generalizations include the introduction of elements that relate to system constructs such as system, module, and component. In this context, the grammar of the programming language being modeled defines the

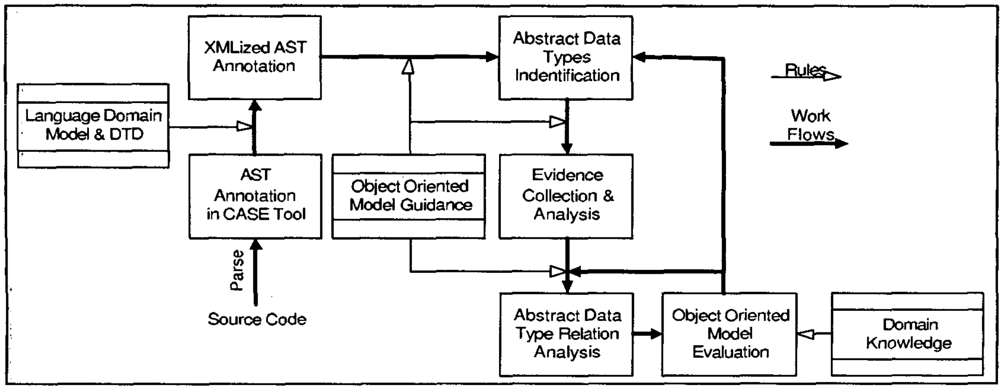

XMLized AST Annotation
Abstract Data Types Indentification
Rules
Work Flows
Language Domain Model & DTD
AST Annotation in CASE Tool
Object Oriented Model Guidance
Evidence Collection & Analysis
Parse
Source Code
Abstract Data Type Relation Analysis
Object Oriented Model Evaluation
Domain Knowledge

Figure 4. Proposed process for migrating procedural systems to object oriented platforms

Document Type Definition (DTD) and consequently the organization of the XML document that models the AST of a given source code fragment. Conformance to the grammar of the programming language being modeled, ensures that the AST DTD specifies the logical structure of the XML element hierarchy and relates to the syntactic entities of the source code. Figure 3 illustrates the AST DTD of the AST are presented in Figure 1.

4. A framework for migration to object oriented platforms

The proposed migration process consists two major phases. The first phase focuses on representing the source code of the system being analyzed at a higher level of abstraction than source text in the form of an annotated Abstract Syntax Tree.

In order to achieve portability, source code is annotated by XML tags, which conform to specific language models. So far, we have developed domain models and XML source code annotators for C, C++, and Java. Once the source code is annotated with XML tags, it can be transferred and imported to an analysis tool as a valid XML document that is compliant to the specific programming language domain model DTD. In this context, the annotated Abstract Syntax Tree of the original system being analyzed is the DOM tree that is generated by parsing the XML document (i.e. the annotated source code). Software analysis and re-engineering tools can be developed on top of such DOM Abstract Syntax Trees. This representation scheme is very portable, since it can be parsed by any XML parser and provides through the DOM tree API an open interface to build reengineering tools. These tools could be fully interoperable since they share the same API as the W3C's DOM tree API.

The second phase aims at the extraction of an object-oriented model by a series of iterative analysis steps applied at the Abstract Syntax Tree level. The focal point in this step is the extraction of classes and the identification of methods that can be attached to these classes. To facilitate this extraction phase, the process focuses on the analysis of global aggregate data types, and formal parameter lists. In this context, global data types and data types in formal parameter lists found in the original legacy source code become primary candidates for classes in the new object oriented system. Similarly, functions and procedures in the original system become primary candidates for methods and are attached to the aforementioned identified classes. During the object model extraction process, many alternative object models can be considered. An evidence model [9] allows for the user to evaluate alternative object models and generate the one that possesses specific quality characteristics such as minimal coupling between classes, maximal cohesion within methods, and the incorporation of inheritance and polymorphism in the extracted model. Other important functions of the evidence model include the method assignment conflict resolution (i.e to assist the user to decide to which class a method is the best to be attached), object model refinement (i.e to assist the user to decide the appropriate aggregations and associations between classes), and quality-based object-oriented source code generation for the new migrant system (i.e. to assist the user to generate readable and maintainable code).

Finally, taking into account of an application domain of the code, the domain knowledge assists to evaluate the extracted object model. The overview of the proposed framework is illustrated in Figure 4.

5. Object oriented model discovery and refinement

In the process of extracting an object-oriented model from procedural source code, it is very important to be able to identify those transformation steps and extraction rules that allow for the generation of a high quality target migrant system. To achieve this goal, we propose a process that consists of specific migration steps namely class creation, method identification, and object model refinement including the identification of associations and aggregations for the new migrant system. In the following sections, we present a catalog of analysis and discovery steps that can be used for extracting a quality object model from procedural systems.

5.1. Class creation

In an object-oriented system, software is structured around data rather than around services of systems. In the search of an object model that can be extracted from procedural source code, we first examine the global data structures and global variables of the procedural system as starting points to identify candidate classes. Based on the usage of data types of formal parameter lists can be collected into one aggregate class to minimize the parameter size of the global variables or data types, methods can be identified and attached to specific classes. Moreover, highly related corresponding method that can be generated from the function or procedure being examined [5].

When more than one design decision is possible, a collection of source code features is used to assist the user to select the object model that optimizes specific metrics and quality characteristics. The extraction steps re discussed in more detail in the sections below.

5.1.1. Private member identification

Data type analysis
A structure is a collection of variables that can be of diverse types. Similar to C++ classes, the structure of a record groups the data items related to each other. In procedural code, for example, C struct and union are constructs that can be used to generate aggregate data structures, indicating a degree of functional and logical coupling among a group of data and functions that are using it. For the migration purposes, each data member of a structure can be converted into the corresponding data field of the class that corresponds to the structure or the record of the procedural data type.

Variable analysis
There are three main scopes to access variables in a program, namely, local scope, file scope and global scope. According to their different scopes, variables can be transformed to private data members of the identified classes. In this context, variables that can only be referenced within a function can be declared as private data members of the class that encapsulates the method that corresponds to the function being transformed.

Although C++ allows for global constant definitions to be accessible from within classes, in order to obtain a better object oriented design for the migrant code, one should aim to eliminate file and global scoped variables that can be referenced from within a file or a whole program. File scope and global scope variables are similar in that they are used or updated by any function. Keeping this scope of variables in the new object oriented system would violate the principles of encapsulation and information hiding. A possible solution is that each of these variables can become a data field of a class that relates to these variables or it encapsulates the functions or part of functions that most often refer or update such variables. Finally, orphan variables that cannot be assigned to any class can be encapsulated in a container class.

5.1.2. Method identification. The method identification process focuses on the discovery of functions from a procedural code that can be transformed into methods in classes. The process of identifying methods and associating them with a class is based on the formal parameter type analysis, return type analysis and variable usage analysis. For this task only aggregate data types are considered, and simple ones such as int, char, and float are ignored.

Parameter type analysis
A formal parameter of a function indicates that the function references a data item of a particular type. In the process of extracting an object model from procedural code, one key analysis is to consider functions of the procedural system as primary candidates for methods in the new object oriented system. These methods can be attached to a class that corresponds to the data type appearing in the formal parameter list. In this context, a function or a procedure may be able to be attached to many different classes when more than one data type exists in the formal parameter list. We then say that this method produces an *assignment conflict.* An evidence model can be used to evaluate the alternative designs and suggest so the user can select the one that optimizes a number of software engineering criteria namely cohesion and coupling of the target system. Parameter type analysis is one of the alternative ways to consider when we would like to decide whether a function should be transformed to a method, or to which a class should be attached. The following points discuss such alternative types of analyses.

Return type analysis
The return type of a function indicates that the function updates variables of a specific type that appears in the return statement. If this return type is also in the formal parameter list, then this provides stronger evidence that

the function should be a method attached to the class stemming from this data type. The reasoning behind this analysis is that classes should encapsulate methods that update the state of objects that belong to this class. Moreover, this type of analysis provides valuable insights as to which class the method should be attached to, when no other information can be extracted from the formal parameter list.

Variable usage analysis
Variable usage analysis allows for examining whether a function or a procedure references a particular data type. Based on the concept of information hiding, we are interested in attaching methods to classes in a way that a given method references as much as possible variables of the same type as the class it is attached to. Variable usage analysis provides such insights and counts as to how many times a variable of a particular class is references within a candidate method. This type of analysis is also of particular importance when no information from formal parameter list and return type analyses could be obtained to decide which class a candidate method should be attached to.

Metrics
Metrics play an important role in deciding to which class a candidate method should be attached. Two particular metrics *Information Flow*, and *Function Point* provide valuable insight on the characteristics of the alternative designs that can occur when considering all the alternative ways of attaching methods to classes. In particular, when an alternative design is evaluated, we are interested on minimizing the Information Flow and the Function Point metrics. The reasoning is that the Information Flow metric provides a measure of the coupling between two different modules (in this case classes). By minimizing this metric, we obtain an object model that has minimal coupling between its classes, which is a highly desirable property. Similarly, the Function Point metric provides a measure of the functionality delivered by a specific module (a class and its methods in this case). For our analysis we would like to minimize the functionality delivered by a class and its methods that is to make it highly cohesive (i.e. delivers a specific well-defined functionality).

Function Splitting
One of the features in legacy procedural code is the size of functions or procedures which tends to increase with prolonged maintenance activities (i.e. when adding new functionality, or when correcting errors). This situation leads to the complex code that is difficult to understand, maintain, and migrate. For the objectification process, such long function or procedure should be sliced into smaller chunks of code. The slicing criteria may vary, but the safest way is to consider slicing according to the first use of a data type that corresponds to a class that the specific function is a candidate to be attached to as a method. Function Point analysis can also be used to confirm that a particular slice provides a highly cohesive piece of code. Finally, source code informal information such as comments and variable names provide also good cues for splitting a function or a procedure into more than one component [5].

5.2. Class association discovery

Inheritance, polymorphism, and overloading, are some of the most important features of an object-oriented design. To achieve a good object-oriented design from a procedural legacy system, we have identified a number of heuristic analysis rules that can be used to establish associations between abstract data types and assist to solve the method assignment conflict problem. Below we discuss the different heuristics to identify inheritance, polymorphism, and overloading opportunities in the extracted object model of the target migrant system.

5.2.1. Inheritance

Data field cloning
If two or more structures differ only with respect to few fields, these can be the candidate subclasses of a more general class. The common fields from those structures are extracted and form a super class. Subclasses can inherit from it with the addition of their own fields.

Data field mapping
Certain functions copy values of the fields from one data structure to the values of the fields in another data structure. These two data structures may not be of the same type, but they share common values. The relation of these data structures can be identified as inheritance, by extracting the common fields into a super class.

Function code cloning
The code clone analysis can identify inheritance where two functions are identical with the only difference that they operate on different data types. In this case, these data types may become subclasses of the more general type and the method can be attached to the subclasses that are inherited from the more general class.

Data type casting
In cast operations, the compiler will automatically change one type of data into another when appropriate. For instance, when an integral value is assigned to a floating-point variable, the compiler will automatically convert the `int` to a `float`. Casting allows to make this type conversion explicit, or to force it when it wouldn't

normally happen. Implicit cast operation between two data types suggests that these data types share common data fields or are interchangeable. The inheritance between these two abstract data types can be deduced in that the casted type becomes the subclass of the type that it is casted to.

Anonymous union type

Anonymous union types denote that their data members share the same memory space. This feature provides a subtle difference from the semantics of a C struct where all members are referenced as single distinct group. By contrast, only one union data member can be referenced at a time, and different data members cannot co-exist in the same time. In this context, the common structure of the union data member can be extracted as a superclass, while each of the union data members can be transformed to a subclass. For example the *Personnel*, a C struct as illustrated in Program 1 below, contains a union type data field. The super-class, Personnel is created and contains the field, age, other than the union data member in the struct illustrated in Program 2, while each of the union data fields becomes a subclass.

```
typedef struct{
       union {
              int student;
              int teacher;
       }
       int age;
} Personnel
```

Program 1. struct definition

```
class Personnel {
       int age;
};

class Student : public Personnel{
       int student;};

class Teacher: public Personnel{
       int teacher;};
```

Program 2. Refactoring ADT into Class inheritance in C++

5.2.2. Polymorphism

Switch statement replacement

One of the most important characteristics of object-oriented designs code is the limited use of switch (or case) statements — Polymorphism provides an elegant way to limit the use of long, complex and cumbersome switch and case statements [5]. A switch statement in the procedural code that uses in its evaluation condition a type check code, can be replaced by polymorphic methods. Specifically, each of the case statement bodies may become a polymorphic method while the data type codes that are used by he condition of the switch or case statement can be considered as candidate dasses in the new migrant system.

Conditional statement replacement

The branch of a conditional statement is executed according to the return value of the evaluating condition. When the type code is used in the conditional expression, then each branch of the conditional statement can be transformed to a polymorphic method, while the type code used by the conditional expresion can be considered as a candidate class in the new design.

```
void printIt(void *itToPrint, int type)
{
       employee *thisEmp;
       market *thisMarket;

       if (type == EMPLOYEE){
              thisEmp = itemToPrint;
              // ......
       }
       if (type == MARKET){
              thisMarket = itemToPrint;
              // ......
       }
}
```

Program 3. Conditional Statement Replacement

Function pointer replacement

There are two ways functions can be invoked h C: by name and by address. Invocation by name is by far the most common one when the functions to be called are decided at the compile time. Invocation by address is used to determine at run time the concrete functions to be executed. In this context, each possible function pointer reference can become a class and their corresponding source code can become a polymorphic method.

Generic pointer parameters replacement

The generic C pointer is denoted by "void *". A "void *" variable can contain the address of any data type. Often this technique is used to write highly generic functions that need to deliver some small piece of non-generic functionality. An example is illustrated in Program 3 below, where the address of struct data type in C, along with a type code is passed into the generic function as a parameter at the printIt() function. At run time, the appropriate struct is accessed by address. In this case, the generic function can be converted into a polymorphic method whereas its behavior is determined according to the type of the object that is applied upon. Moreover, the type codes that can be referenced by the generic pointer parameter can be transformed to classes in the new migrant system.

5.2.3. Overloading

Source code cloning
When two or more functions are identified as clones with minor differences in their structure and the data types they use, these functions can be overloaded on the data types they differ. The constraint is that these functions should return the same data type.

Functions with common prefix or suffix name
Similar prefix and suffix names in functions or procedures provide important cues for overloading. For example, execl(), execv(), execlp(), execp() are functions to execute unix processes in various ways and all can be overloaded according to the type they operate upon.

Functions with union type parameter
The functions with a union type parameter may become candidates for overloaded methods. The reason is that these functions usually have different behavior according to the type of the union parameter they are applied upon. These functions can be transformed into several overloaded methods with different parameter types that are obtained from the original union structure definition. Each overloaded method can operate on the specific case of the original union structure.

6. Experiments

To investigate the usefulness of the migration process and heuristics presented in this paper, three different systems (AVL tree libraries, bash, tcsh) have been annotated by XML tags and represented by DOM Abstract Syntax Trees. A fourth system (WELTAB) has been analyzed and migrated to an object oriented platform, from its original C implementation.

6.1. Source code representation in XML

The domain model for the C programming language was examined and C source code for various systems has been represented in the form of an XML document and an XML DOM tree.

We have used the Refine/C parser by Reasoning to obtain an XML version of the C source code. In this context, we could have used any parser for this task. We have chosen the Refine parser because of the flexibility of the API it offers and the rich domain model that defines in the form of language entities taxonomy. For example, in the C domain model we denote that `multiplication` language construct is a subclass of `arithmetic-expression`, which is in turn a subclass of the `expression` construct. All language constructs can be represented as classes in the language domain model. Moreover, the domain model defines the attributes of these object classes in terms of mappings from one object class to another. The domain model also specifies which of these attributes define the structure of a C Abstract Syntax Tree and which attributes define annotations on the tree [1].

System	*Size of Source Code*	*Size of AST XML*
AVL Library	164,401bytes	1,660,167 bytes
Bash	628,919 bytes	25,421,443 bytes
Tcsh	930,644 bytes	47, 444,861bytes

Table 1. Comparison between the Size of Source Code and AST XML Document

Based on ANSI C grammar and leveraging the concepts in C domain mo del [1], we have created an AST XML structure that was specified in terms of a DTD. In this context, Figure 3 illustrates part of the resulting DTD. For this work we have constructed the XML DTD manually, by examining the C language model. However, on-going work in our group is focusing on extracting such a DTD automatically using the IBM Eclipse framework, the MOF XMI, and a UML description of the language domain model. Table 1 provides some comparison statistics related to the size of the original source codes and the size of the generated XML documents. A detailed discussion on representing source code as XML documents can be found in [11].

6.2. Object oriented model discovery

For our experiments we have applied the proposed objectification technique to extract an object model from the WELTAB system. WELTAB was created in 1970 to support collection, reporting and certification of US federal election results. The system is consisted of a set of C program and common data files. In total it has 190 files, including 39 C source files, 26 Library files, 20 driver files and rest of data files.

The object model identification process as discussed in the previous sections is primarily based on data type analysis, global variables, formal parameter analysis and, data usage. First, all data types of global variables declared in a file are collected. Then parameter lists are examined and the corresponding data types are also collected. All the collected data types become candidate classes. For example, as it is illustrated in Program 4 below, the global variables in file state.h are grouped together to form the private members of a class named `OFFICE`.

```
long int form[6][21];
int nform;
int outpos;
```

Program 4: Global Variables

The functions fetching or storing variables related to the collected data types become primary candidates for methods. A data type, along with the set of functions operating on this data type form an object. Files, which provide formatted input or output, are examined, and their ftructure is used for orming classes. Private data members can be obtained from local variables in functions or fields in structures.

In Figure 5, the object model obtained from the original C source code of the WELTAB system is illustrated. The obtained object model is presented using UML and can be refined by the software engineer using the UML editor from the tool Together^{TM}. The main unit for the migrated WELTAB system is the class WELTAB, which is associated with the classes RECORD, OFFICE and VOTEUNIT. Finally, the migrant C++ source code is automatically generated, by using the extracted object model and the DOM Abstract Syntax Tree. Some automatically generated sample code is illustrated in Figure 6.

7. Conclusion

In this paper, we presented a framework to migrate systems written in procedural code into an object-oriented platform. The software analysis is based on a portable XML source code representation that utilizes a domain model and consequently maps the entities of the programming language domain model to a DTD. The XML document effectively corresponds to an AST and has a standard structure and API. We consider that such an XML based representation of the source code can be thought of as a first step towards CASE tool interoperability.

Moreover, we presented a process and a list of heuristic steps that allow for a quality object model to be extracted from the legacy source. The proposed process is

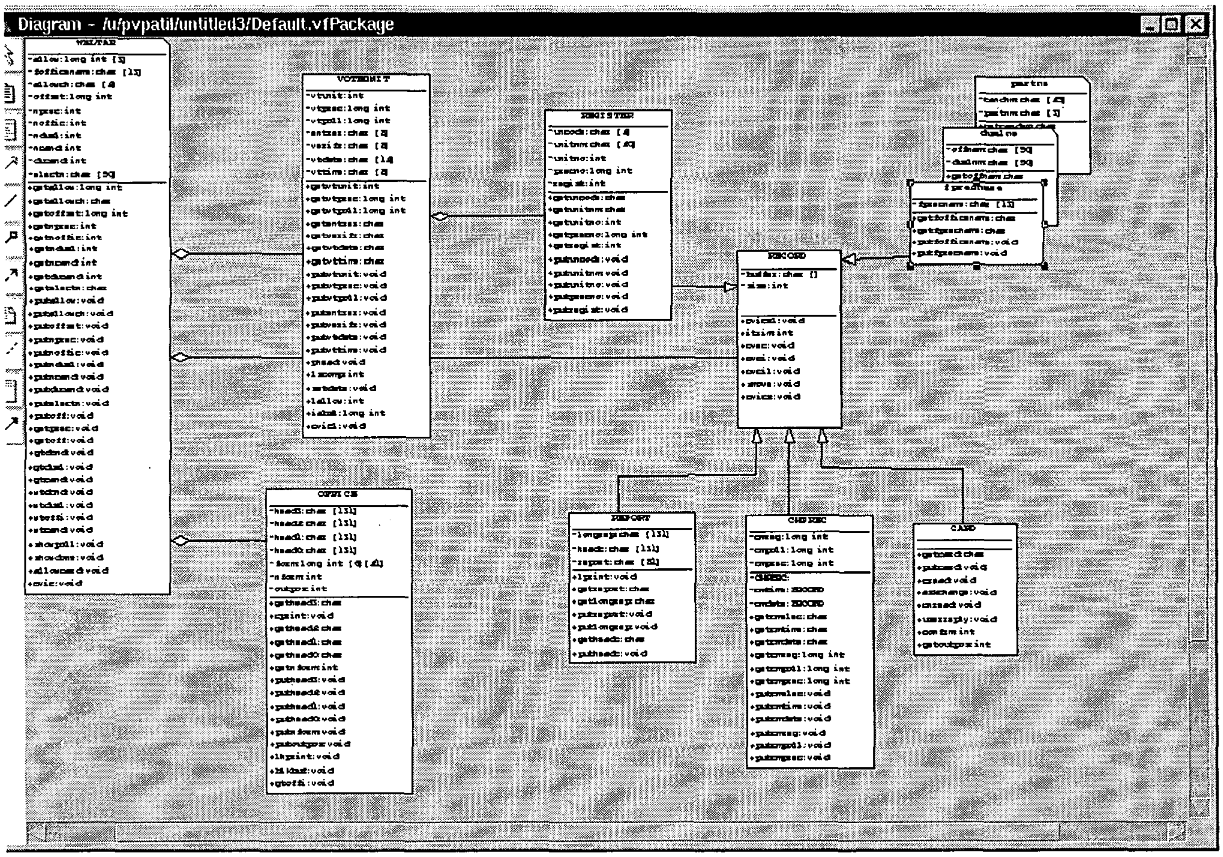
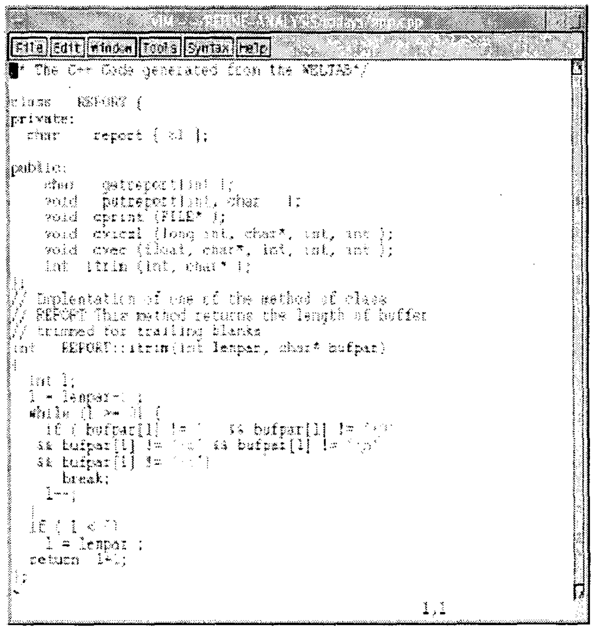


Figure 5. Object Model in UML Diagram



```
/* The C++ Code generated from the RELTAB*/

class   REPORT {
private:
  char    report [ 81 ];

public:
    char   getreport(int );
    void   putreport(int, char   );
    void  cprint (FILE* );
    void  cvtcal (long int, char*, int, int );
    void  cvec (float, char*, int, int, int );
    int  ltrim (int, char* );
};
// Implentation of one of the method of class
// REPORT This method returns the length of buffer
// trimmed for trailing blanks
int   REPORT::ltrim(int lenpar, char* bufpar)
{
  int l;
  l = lenpar-1 ;
  while (l >= 0) {
    if ( bufpar[l] != ' '  && bufpar[l] != '\0'
   && bufpar[l] != '\n' && bufpar[l] != '\t'
   && bufpar[l] != '\r')
      break;
    l--;
  }
  if ( l < 0)
    l = lenpar ;
  return  l+1;
};
```

Figure 6: Sample Source Code Generated from Object Model Discovery

incremental and iterative up to the point that the user is satisfied with the quality characteristics of extracted object model. The quality characteristics we consider in this work focus on the minimization of coupling and the maximization of cohesion in the derived object model. The proposed technique has been applied for the migration of various C systems to C++ at the IBM Center for Advanced Studies and it is shown to be scalable and extensible. Moreover, by following such a migration framework, reusable components can be identified and be wrapped using CORBA or SOAP wrappers so that, they can be invoked as Web services in Web-enabled environments.

Future extensions on the work presented in this paper may focus on two directions. The first direction is on investigating the use of the XMI MOF model in order to obtain a standardized domain model DTD for a given programming language. The second direction is on extending the heuristic rules in order to obtain a more refined object model. his work is conducted in collaboration with IBM Center for Advanced Studies at the IBM Toronto Lab.

8. References

[1] Reasoning Systems, "Refine/C Programming's Guide", June 1995.

[2] Letha H. Etzkorn, Carl G. Davis, "Automatically Identifying Reusable OO egacy Code", Computer, IEEE, October, 1997.

[3] Michael W. Godfrey, "Defining, Transforming, and Exchanging High-Level Schemas", th WCRE'2000.

[4] De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli, "Migrating Legacy Systems toward Object-Oriented Platforms", 1997, IEEE.

[5] Martin Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 2000.

[6] R. C. Holt, et al., "GXL: Toward a Standard Exchange Format", 7th WCRE 2000.

[7] R. Koschke, J.-F. Girard, and M. Würthner, "An intermediate representation for integrating reverse engineering analyses" , 7th WCRE'2000.

[8] E. Mamas, K. Kontogiannis, "Towards Portable Source Code Representation Using XML", 7th WCRE'2000.

[9] P. Patil, Y. Zou, K.Kontogiannis, J. Mylopoulos, "Migration of Procedural Systems to Network-Centric Platforms", CASCON 1999.

[10] Jörg Czeramski, et al, "Data Exchange in Bauhaus",. 7th WCRE'2000.

[11] http://www.swen.uwaterloo.ca/~yzou/AST/.

[12] K. Kontogiannis, P. Patil, "Evidence Driven Object Identification in Procedural Systems". STEP'99, September 1999, pp. 12-21.