Incremental Transformation of Procedural Systems 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

Over the past years, the reengineering of legacy software systems to object oriented platforms has received significant attention. In this paper, we present a generic re-engineering source code transformation framework to support the incremental migration of such procedural legacy systems to object oriented platforms. First, a source code representation framework that uses a generic domain model for procedural languages allows for the representation of Abstract Syntax Trees as XML documents. Second, a set of transformations allow for the identification of object models in specific parts of the legacy source code. In this way, the migration process is applied incrementally on different parts of the system. A clustering technique is used to decompose a program into a set of smaller components that are suitable for the incremental migration process. Finally, the migration process gradually composes the object models obtained at every stage to generate an amalgamated object model for the whole system. . A case study for the migration of a medium size C system to C++ is discussed as a proof of concept.

1. Introduction

Legacy systems are mission critical software systems that entail comprehensive business knowledge and they constitute large assets for organizations. However, their quality and operational life are constantly deteriorating due to the maintenance activities. With the rapid technology updates, there is great pressure to migrate or port existing systems into modern platforms where better and faster operating, development, and maintenance environments exist. One possible solution to leverage the business value of such systems is to re-engineer them into the object-oriented platforms. With properties such as encapsulation, inheritance, and polymorphism inherent in object-oriented designs, the migrated systems can be easier maintained, reused and integrated with other applications in network centric environments.

This work is supported by the IBM Center for Advanced Studies IBM Toronto Laboratory, NSERC, and the Consortium for Software Engineering Research, Canada.

Most of today's legacy systems are written in procedural languages. In a nutshell, the object oriented migration process involves the analysis of the Abstract Syntax Tree (AST) of the procedural code, the identification of object models, and the generation of object oriented code with desired software quality levels. In this context, the software reverse engineering community has proposed methods to migrate systems written in various procedural languages, such as COBOL[6], Fortran[11], and C[7, 8], into object-oriented platforms. In this paper, we propose an incremental source code transformation framework that allows for procedural system to be migrated to modern object oriented platforms. First the system is parsed and a high level model of the source code is extracted. In the proposed framework we introduce the concept of a unified domain model for a variety of procedural languages such as C, Pascal, COBOL, and Fortran. Such unified models can be implemented in XML and denote common language features such as routines, subroutines, function, procedures, types, statements, variables and declarations, just to name a few. Second, to keep the complexity and the risk of the migration process into manageable levels, a clustering technique allows for the decomposition of large systems into smaller manageable units. A set of source code transformations allows for the identification of an object model from each such unit. Finally, an incremental merging process allows for the amalgamation of the different partial object models into an aggregate composite model for the whole system. In this way, the migration task is tackled in a "divide-conquer" manner. The sections below discuss these concepts in detail.

This paper is organized as follows. Section 2 discusses the concepts pertaining to a unified domain model for the representation of procedural code using XML formats. Section 3 presents an incremental transformation process to migrate procedural systems to object oriented platforms. Section 4 presents case studies that utilized the proposed approach. Finally section 5 concludes the paper.

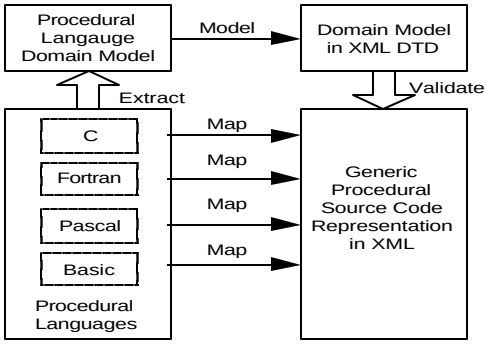

Figure 1: Generic Procedural Code Representation Framework

2. Source Code Representation

In order to analyze the source code, it is critical to represent the program source code at a higher level of abstraction than source code text. Program representation provides means to generate abstractions, appropriate input to a computational model for analyzing and reasoning about programs, and methods for the translation and normalization of programs. Various high level source code representation techniques are presented in [1, 2, 3, 4, 5]. In this section, we discuss the program representation techniques that are based on procedural language domain models and the XML markup language.

2.1. Domain Model for Procedural Languages

To build a generic representation for specific categories of procedural languages, there are two major steps involved, namely, the abstraction of the individual procedural language domain models and the representation of such an abstraction in a generic format, as illustrated in Figure 1. The focal point is to identify the functional equivalent constructs and aggregate them at a higher level of abstraction. For example, for the purpose of language modeling, language constructs, such as “subroutine” in Fortran and “function” in C, denote similar concepts that can be aggregated by a unique term “procedure”. Due to the size limitations in this paper, we will focus our discussion on a domain model that stems mostly from the C programming language. The specific domain model is generic enough to handle constructs from various programming procedural languages, such as Fortran, Pascal, and COBOL. For this purpose, the UML object-oriented modeling language is utilized to denote language syntactic constructs in AST (Abstract Syntax Tree) nodes as UML classes and AST edges as class associations. The association links represent the attributes of non-primitive language syntactic types and are denoted as mappings from one UML class to another. A subset of such language domain model for Expressions represented in UML class diagram is illustrated in Figure 2. As shown in

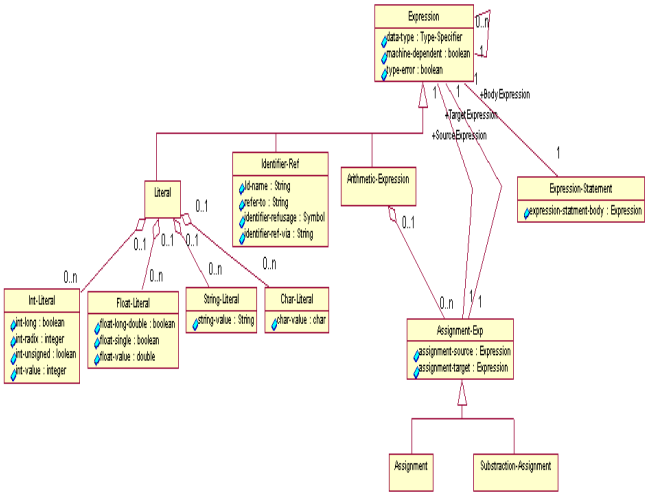

Figure 2: Sample Domain Model for Expressions

Figure 2, `expression` is a basic construct, including sub-classes, such as `identifier reference`, `literal constant`, `arithmetic expression`, or `assignment expression`. The `expression`, as a base class, contains the common attributes and shares them with its subclasses, such as `literal`, `arithmetic expression`, `identifier reference` by inheritance. The `expression` is self-reflective, with which an association points back to itself. Therefore, an `expression` can contain one or more `expressions`.

2.2. Representation of ASTs in XML

The Abstract Syntax Tree (AST) provides the most detailed information about a source code. There are two approaches to extract the abstract syntax tree 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, and C. Tools that utilize this approach include Refine for C/Fortran/COBOL, Datrix for C++/C/Java.
The other 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 language grammar rules, different parsers can extract the necessary information from the source code and encode it in a uniform and language-neutral format. Using a domain model definition extracted from the specification of a given programming language (i.e. ANSI C), a hybrid approach to define the logical structure of the entities of an Abstract Syntax Tree in terms of a Document Type Definition (DTD) document can be utilized by following the steps below.

In the first step, a domain model for a given programming language is defined as a collection of

```
<EXPRESSION-STATEMENT >
  <EXPRESSION-STATEMENT-BODY>
   <EXPRESSION>
    <ASSIGNMENT-EXP>
     <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" int-value="26"/>
         </MULTIPLICATION-ARGS>
       </MULTIPLICATION>
      </ASSIGNMENT-SOURCE>
     </ASSIGNMENT>
    </ASSIGNMENT-EXP>
   </EXPRESSION>
  </EXPRESSION-STATEMENT-BODY>
</EXPRESSION-STATEMENT>
```

Figure 3: XML Element Structure for Expression Statement in C

classes, hierarchies, and association. By recursively traversing the hierarchy of the domain model entities, the given domain model can be mapped to a Document Type Definition (DTD). Specifically, domain model classes are mapped as DTD elements, and associations are mapped as DTD attributes. During parsing the semantic actions of the parser can be used to generate an XML representation of the source code as illustrated in Figure 3. In the sec ond step, 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 generic XML based representation for procedural code can be designed as to contain common language structures found in a group of programming languages including files, libraries, data types, data definitions, variables, constants, macros, expressions, statements, I/O utilities and functions. For our work, the AST of each individual procedural language is extracted and represented in the XML format. The XSLT (eXtensible Stylesheet Language transformation) is used to define transformation rules to generalize individual domain models represented as a DTD to more generic domain models. An example mapping for Fortran constructs is illustrated in Table 1.

3. Incremental Migration Process

In this section, an incremental process model to migrate legacy systems into object-oriented platforms is presented. The need for an incremental migration process is strong since large systems are not migrated at once because of the complexity and the risk involved. It is therefore important that a technique that identifies system segments that serve as "work areas" in the migration

Fortran Domain Model	***Generalized Domain Model***
Structure-Statement	Structure
Record-Statement	Struct Variable Declaration
Common-Statement	Global Variable Declaration
Programs	Program
Executable Program	File
Program Unit	Function-Def
Type-Statement	Declaration
Read-Statement	Function-Call
Call-Statement	Function-Call
Indexable-Name/function-Params-	Function-call
Character-Statement	String
Equivalence-Statement	Union-Struct
Intrinsic-Statement	Function-Pointer

Table 1: Generalization of the Fortran Domain Model

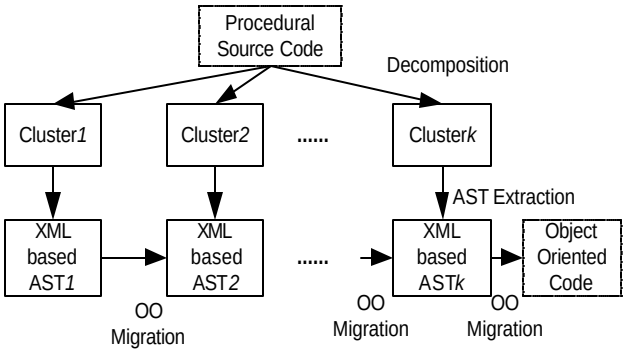

Figure 4: Incremental Object Oriented Migration Process

process to be devised. We aim to divide a system into a collection of cohesive "work areas", which group exclusively related entities for the migration process. Consequently, the XML based AST for every such "work-area" segment is generated, and in turn the migration process operates iteratively on each segment (shown in Figure 4). In the following subsections, these issues are addressed in detail.

3.1. A System Segmentation Algorithm

Most clustering techniques presented in literature utilize certain criteria to decompose a system into a set of meaningful modular clusters. Such criteria attempt to achieve a cluster with low coupling, high cohesion, interface minimization and shared neighbors. In the context of the object-oriented migration, we strive to produce clusters that assemble the maximum source code properties related to a class candidate. In this respect, essential source code entities are called *seeds.* Other *entities* that *associate* with this *seed* entity form a cluster. An *association* is a directed edge from a *seed* to its related *entities*.

Criteria on the Selection of a *Seed*

A *seed* is selected according to its potential to be considered a class candidate in the new migrant system.

In this context, a *seed* can be chosen from aggregate data types, global variable declarations, and function pointer declarations. Specifically, the aggregate data types include *struct* type definitions, *union* type definitions, *arrays*, and *enumeration* definitions. In this case, the fields in an aggregate data type become the data members in a class candidate. Similarly, a global variable is encapsulated as a data member in a class candidate. Moreover, a function pointer declaration is treated as a clustering *seed* for the reason that a function pointer declaration defines a type for the functions passed as parameters.

Criteria on the Selection of *Entities*

Due to the object oriented design principle that a class encapsulates data and the related methods, we focus on the discovery relations between data declarations and functions that use such data. Such relations include type references, data updates, and data uses. The algorithm for selection of *entities* given a *seed* is illustrated in Program 1.

Furthermore, the clustering algorithm, illustrated in Program 2, is composed of three major steps. First, all the functions in the original procedural system are identified and stored in a set F. Second, all the *seed* candidates are identified and stored in a sequence. Finally, for each *seed* the associated *entities*, including functions and aggregate data types are collected in a cluster. Consequently, every result cluster is represented as a tuple in the form of a tuple *<seed, associated function set, associated aggregated data type set>*.

The overall clustering algorithm takes the AST(Abstract Syntax Tree) as an input, and produces clusters represented by a sequence of tuples as an output. The pseudo code programs in below illustrate the process for identifying migration work-area segments.

Program 1: Algorithm for Collecting Related Entities

Collect_Related_Entities(T_i, S)
Input:
T_i: an aggregate type, a global variable, or function pointer type considered to be a seed
S: AST view of a system
Output:
P: a tuple contains a seed, T_i, a set of related functions, M_{T_i}, and a set of related aggregated types, R_{T_i}
Algorithm:
Begin:
--Initialize the set of related functions
$M_{T_i} = \varnothing$;
--Initialize the set of related aggregated types
$R_{T_i} = \varnothing$;
-- T_j has a data member with the type of T_i
$R_{included\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } hasDataMember\_ber(T_j) = T_i\}$;
-- T_j is casted into the type of T_i
$R_{casted\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } isCastedTo\_(T_j) = T_i\}$;
--T_j and T_i have common data members
$T_{cloned\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } hasDataMem\_bers(T_j) \subseteq getDataMem\_bers(T_i)\}$;
-- T_j and T_i have data members that are mapping to each other
$T_{mapped\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } hasDataMem\_ber(T_j) = hasDataMem\_ber(T_i)\}$;
$R_{T_i} = T_{included\_types} \cup T_{casted\_types} \cup T_{cloned\_types} \cup T_{mapped\_types}$;
-- F_j has parameters with the type of T_i
$M_{parameter} = \{F_j \mid F_j \in F \text{ and } hasParameter(F_j) = T_i\}$
-- F_j has return value with the type of T_i
$M_{return} = \{F_j \mid F_j \in F \text{ and } hasReturn(F_j) = T_i\}$
-- F_j update variables related to T_i
$M_{update} = \{F_j \mid F_j \in F \text{ and } T_i \in updatedTypes(F_j)\}$
-- Fj is the actual passing function to the function pointer type
-- parameter
$M_{actual\_functions} = \{F_j \mid F_j \in F \text{ and } isFunctionPointerDeclaration(T_i) \text{ and } getType(F_j) = T_i\}$
$M_{T_i} = M_{parameter} \cup M_{return} \cup M_{update} \cup M_{actual\_functions}$
P←<T_i, M_{T_i}, R_{T_i}>;
return P
End

Program 2: Algorithm for System Segmentation

Segment_System (S)
Input:
S: AST view of a system
Output:
S_p: set of clusters
Algorithm:
Begin
-- Initialize a cluster P into an empty tuple
P= ∅;
-- Initialize the cluster set S_p into an empty set
S_p=∅;
-- Identify all the functions and store them in a set F
F ← getAllFunctions(S);
--Identify all seed candidates
[T_1, T_2, …, T_n] ← getAllSeeds(S);
T ← [T_1, T_2, …, T_n];
-- Identify clusters
for each type T_i in T **loop**
$< T_i, \{F^1_{T_i}, ..., F^k_{T_i}\}, \{T^1_{T_1}, ..., T^m_{T_i}\} > \leftarrow$ **Collect_Related_Entities**(T_i);

$P \leftarrow < T_i, \{F_{T_i}^1, ..., F_{T_i}^k\}, \{T_{T_1}^1, ..., T_{T_i}^m\} >;$

$S_p \leftarrow S_p$ with P;

end loop;

$S_p = [P_1, P_2, \ldots P_n]$;

return S_p

End

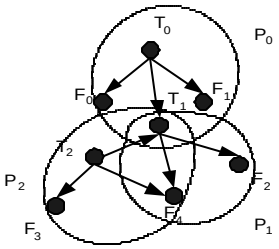

Figure 5: Example for System Segmentation Algorithm

Figure 5, illustrates a result of applying the clustering algorithm, where three clusters have been identified:

$P_0 = <T_0, \{F_0, F_1\}, \{T_1\}>$,

$P_1 = <T_1, \{F_2, F_4\}, Æ>$,

$P_2 = <T_2, \{F_3, F_4\}, \{T_1\}>$.

The algorithm and the clustering criteria allow for overlapping regions to exist. When the overlap occurs on aggregate data types, it may indicates a multi-inheritance relationship among the generated class candidates. However, if an overlap occurs on functions it reflects conflicts in method assignment. As one function can be only attached to one class as a method, the conflict has to be resolved. In order to achieve good quality in the migrated system, we provide a qualitative method and an evidence model to determine the choice of the class candidate that the function should be attached to [9, 10]. Finally, in the context of incremental migration process, it is important to independently select and migrate a cluster without relying on the information in other clusters. In this respect, the shared regions are duplicated in each cluster while converting the clusters in the XML format. In other cases that some functions are not related to any seeds, we wrap such "leftover" functions into one cluster.

3.2. Incremental Migration Process

The decomposition of a program produces a set of smaller work areas. The algorithm for the incremental transformation is illustrated in Program 3. It takes the sequence of identified clusters as input and generates incrementally an object-oriented system in the end. The migration process is divided into *k* phases one for each cluster. The algorithm iterates over each cluster, and updates the system object model, referred to as OM. It is worth to note that the clusters are applied in the order that is constrained by the function calls inside the shared functions. As previously stated, the shared functions will cause the conflicts in method assignment. The conflicted functions called by other functions in conflict should be resolved first. Therefore, the clusters with less function call dependencies in the shared functions are migrated first. In summary, the transformations are performed in four steps, (as shown in Program 4):

These steps are:

1) generate new class candidate which is added into the object model;
2) attach the associated functions into the new class candidate, and update the object model;
3) resolve conflicts when a function can be assigned to either the current class candidate or the existing class candidates in the object model (the resolving techniques have been presented in the research papers [9, 10]); and finally,
4) refine the object model by identifying class inheritance.

A comprehensive set of transformation rules to perform each of above migration steps are presented in [9, 10]. To govern the order of transformation composition, the pre/post conditions for each transformation are formally specified in OCL.

Program 3: Algorithm for Incremental Transformation

Transform_Clusters (S_p)
Input:
-- a sequence of clusters, k is the number of the clusters
$S_p=[P_1,P_2,\ldots,P_k]$
Output:
OM: object model of the system
Algorithm:
Begin:
OM=∅;
Order_Clusters(S_p);
while (phase < k) **do**
OM_{phase}=**Generate_Object_Model**(OM, P_{phase});
OM= OM_{phase};
phase = phase + 1;
end while
return OM
End

Program 4: Algorithm for Generating Object Model from a Cluster

Generate_Object_Model(OM_{i-1}, P_i)
Input:
OM_{i-1}: the accumulated object model from clusters 1..i-1;
P_i: the *i*th cluster;
Output:
OM: the accumulated object model from clusters 1..i;
Algorithm:
Begin:
OM=**Generate_Class**(OM_{i-1}, P_i);
OM=**Attach_Methods**(OM, P_i);
OM=**Resolve_Conflicts**(OM, P_i);
OM=**Refine_Object_Model**(OM);
return OM;
End

System	*C Source Code Size*	*Cluster #*	*System Size in XML*	*Avg. Cluster Size in XML*
AVL Tree	168,286 Bytes	9	1.69MB	278,722 Bytes
CLIPS	983,127 Bytes	325	38.03MB	423,084 Bytes
BASH	1,257,838Bytes	327	16.9MB	404,992 Bytes

Table 2: System Segmentation Result

4. Case Studies

To investigate the effectiveness of the generic procedural source code representation framework, the domain model for the C programming language was examined and generalized. Furthermore, the C source code for various systems has been represented in the form of XML DOM trees. 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, but we have chosen the Refine parser because of the flexibility of its API. Table 2 provides some comparison statistics related to the size of the original source codes, the number of clusters identified, the size of the original system in XML format, and the average size of a cluster in XML format. As shown in the Table 2, each component is of a manageable size for the software analysis purposes.

Furthermore, we have applied the proposed incremental migration technique to extract an object model for the BASH (Bourn Again SHell) originally written in C. A software analysis tool that is based on the proposed migration process was developed to migrate it into C++. For this case study, we have identified 186 classes including 91 classes generated from the aggregate data types as seeds and 95 classes from global variable declarations as seeds. Finally, a partial of the class diagram for the migrated system is illustrated in Figure 6.

5. Conclusion

In this paper, a unified source code representation framework that utilizes language domain models is represented in XML and Data Type Definition documents. The focal point is to select a model that is rich enough to express all the possible syntactic constructs in the procedural languages. To facilitate the analysis of large systems, a system segmentation algorithm is provided to decompose a program into a set of smaller components where incremental migration can be achieved. In such a way, a large system can be reengineered gradually in order to reduce the risk and computation costs involved.

Future work will include the design of wrappers that allow for the integration of system components that have been already migrated to an object oriented platform with

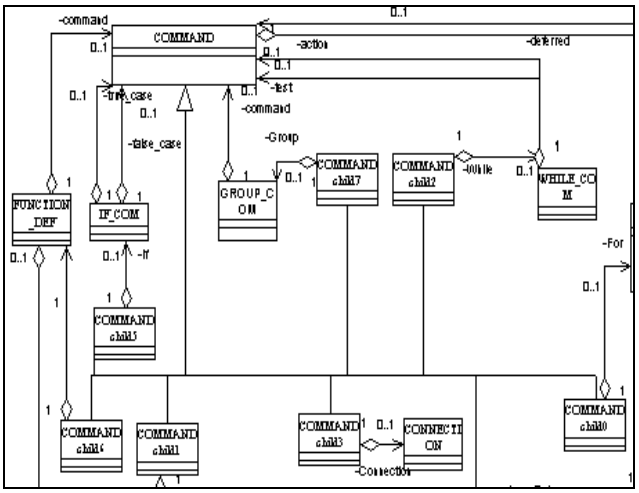

Figure 6: Class Diagram of Migrated System

the rest of the legacy system that is still is in its original procedural form.

References

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

[2] Greg Badros, "JavaML: A Markup Language for Java Source Code", http://www.cs.washington.edu/homes/gjb/papers/javaml/javaml.html.

[3] R. Koschke, J.-F. Girard, and M. Würthner, "An intermediate representation for integrating reverse engineering analyses" , 5th WCRE'2000, November 2000.

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

[5] http://swag.uwaterloo.ca/~dean/cppx/

[6] Harry Sneed, "Object Oriented COBOL Recycling", in the Proceedings of 3rd Working Conference of Reverse Engineering, 1996.

[7] Ying Zou, Kostas Kontogiannis, "A Framework for Migrating Procedural Code to Object Oriented Platform", in the proceedings of 8th Asia-Pacific Software Engineering Conference, 2001.

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

[9] Ying Zou, Kostas Kontogiannis, "Quality Driven Transformation Compositions for Object Oriented Migration", the 9th IEEE Asia Pacific Software Engineering Conference (APSEC), Gold Cost, Queensland, Australia, December 2002 , pp. 346-355.

[10] Ying Zou, Kostas Kontogiannis, "Migration to Object Oriented Platforms: A State Transformation Approach", the 19th IEEE International Conference on Software Maintenance (ICSM), Montreal, Quebec, Canada, October 2002, pp.530-539.

[11] Theurich, G.; Anson, B.; Hill, N.A.; Hill, *A.;* "Making the Fortran-to-C transition: how painful is it really?" Computing in Science & Engineering, Volume: 3 Issue:1, Jan/Feb 2001 Page(s): 21 –27.