Acta Universitatis Apulensis,
ISSN 1582-5329, 2011,
Special Issue on Understanding Complex Systems,
Eds. Barna Iantovics, Ladislav Hluchý and Roumen Kountchev

UNSUPERVISED TRANSFORMATION OF PROCEDURAL PROGRAMS TO OBJECT-ORIENTED DESIGN

Istvan Gergely Czibula and Gabriela Czibula

Abstract. Object-oriented programming has many advantages over conventional procedural programming languages for constructing highly flexible, adaptable, and extensible systems. Therefore a transformation of procedural programs to object-oriented architectures becomes an important process to enhance the reuse of procedural programs. Moreover, it would be useful to assist by automatic methods the software developers in transforming procedural code into an equivalent object-oriented one. In this paper we aim at introducing an agglomerative hierarchical clustering algorithm that can be used for assisting software developers in the process of transforming procedural code into an object-oriented architecture. We also provide a code example showing how our approach works, emphasizing, this way, the potential of our proposal.

Keywords: *software engineering, procedural systems, object-oriented systems, machine learning, clustering*

2000 *Mathematics Subject Classification*: 68N30, 62H30.

1. Introduction

Object-oriented programming has many advantages over conventional procedural programming languages for constructing highly flexible, adaptable, and extensible systems [16]. It is well known that software evolution is an inevitable process for software systems. Repeated changes alter the structure of a system, rapidly degrading it and making the system "legacy". Reengineering seems to be a promising approach to upgrade these systems according to the latest technologies [1].

Object-oriented concepts are useful concerning the reuse of existing software. Therefore a transformation of procedural programs to object-oriented architectures becomes an important process to enhance the reuse of procedural programs. The evolution of some legacy software systems often requires the rewriting of the system into an object-oriented programming language. This activity, especially for large software systems, is difficult and time consuming. That is why it would be useful to assist by automatic methods the software developers in transforming procedural code into an equivalent object-oriented one.

Unsupervised classification, or clustering, as it is more often referred as, is a data mining activity that aims to differentiate groups (classes or clusters) inside a given set of objects [2], being considered the most important *unsupervised learning* problem.

The resulting subsets or groups, distinct and non-empty, are to be built so that the objects within each cluster are more closely related to one another than objects assigned to different clusters. Central to the clustering process is the notion of degree of similarity (or dissimilarity) between the objects.

We have previously introduced in [3] a clustering approach for transforming procedural systems into object-oriented ones. For this purpose, a partitional clustering algorithm, named *kOOS*, was introduced.

In this paper we aim at extending the approach from [3] by introducing a *hierarchical agglomerative* clustering algorithm that can be used for re-grouping the entities from an existing software system written in a procedural language. The goal is to obtain a partition of a software system, in which each cluster would correspond to an application class from the equivalent object-oriented system.

The rest of the paper is structured as follows. The clustering approach for assisting developers in the process of transforming software systems written

in procedural programming languages into object-oriented systems that we have previously introduced in [3] is described in Section 2. Section 3 presents some existing work related to the considered problem. A hierarchical clustering algorithm for transforming procedural systems into object-oriented ones is introduced in Section 4. Section 5 presents an experimental evaluation of our approach and Section 6 provides a comparison of our approach with similar existing ones. Some conclusions of the paper and further research directions are outlined in Section 7.

2. A clustering approach for OO transformation. Background

We have previously introduced in [3] a clustering approach for transforming procedural systems into object-oriented ones. In the following we will briefly describe the proposed approach.

Let $S = \{s_1, s_2, ..., s_n\}$ be a non object-oriented software system, where $s_i, 1 \leq i \leq n$ can be a subprogram (function or procedure), a global variable, a user defined type.

In the following we will refer an element $s \in S$ as an *entity*.

In order to transform S into an object-oriented system, we have proposed an approach consisting of two steps [3]:

- **Data collection** - The existing software system is analyzed in order to extract from it the relevant entities: subprograms, local and global variables, subprograms parameters, subprograms invocations, data types and modules, source files or other structures used for organizing the procedural code.
- **Grouping** - The set of entities extracted at the previous step are grouped in clusters. The goal of this step is to obtain clusters corresponding to the application classes of the software system S.

In the **Grouping** step we propose a *hierarchical agglomerative* clustering algorithm, *HOOS* (*H*ierarchical Agglomerative Clustering for *OO* Transformation).

3. Related Work

In this section we present some existing work in the field of transforming procedural code into object-oriented systems.

A generic re-engineering source code transformation framework to support the incremental migration of procedural legacy systems to object-oriented platforms is presented in [4]. 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 incrementally applied on different parts of the system. A partitioning algorithm 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 object model for the whole system.

The approach from [4] aims to discover relations between data declarations and functions that use such data starting from a set of initial seeds consisting of all aggregate data types, all global variable declarations, and all function pointer declarations.

The paper [1] describes a tool to reengineer procedural systems written in Cobol, Fortran, C or Pascal, into object-oriented ones written in Smalltalk. The developed prototype automatically identifies potential classes, but allows user intervention to work up conflicts.

The paper [5] shows how an object-oriented development method can be used to gradually modernize an old system, i.e re-engineer the system. The proposed technique is based on experiences from real projects.

The approaches from [1, 5] describe re-engineering processes for transforming procedural systems into object-oriented ones, but without focusing on automating the transformation process.

A reengineering tool for automatically transforming a system composed of procedural programs into a functionally comparable object-oriented system is introduced in [16]. The transformation into the object-oriented form locates redundant, duplicated and similar data and processes and abstracts them into classes and methods.

A program transformation process, which transforms originally procedural systems to object-oriented systems is described in [7]. The objects of the resulting system may then be used for further object-oriented systems engineering,

avoiding many problems arising in connection with procedural software reuse, such as module interconnection. This approach focuses on the problem of software reuse.

Scenarios and strategies for gradually reengineering into object-oriented are described in [8]. Techniques for encapsulation of legacy systems into object wrappers are discussed in [9]. Techniques, but not automation, for abstracting of functions from COBOL programs are described in [10]. Analytical analyses with partial automation of the process for abstracting objects and object-oriented specification from procedurally oriented programs are reported in [11], and highly automated techniques for recovering abstract data types and object instances for C are discussed in [12].

4. A Hierarchical Clustering Algorithm for OO Transformation

In our clustering approach, the objects to be clustered are the entities from the software system S, i.e., $\mathcal{O} = \{s_1, s_2, \ldots, s_n\}$. Our focus is to group similar entities from S in order to obtain groups (clusters) that will represent classes in the equivalent object-oriented version of the software system S.

In order to express the dissimilarity degree between the entities from the software system S, we will use an adapted generic cohesion measure [13]. Consequently, the distance $d(s_i, s_j)$ between two entities s_i and s_j is expressed as in Equation (1).

$$d(s_i, s_j) = \begin{cases} 0 & if \quad i = j \\ 1 - \frac{|prop(s_i) \cap prop(s_j)|}{|prop(s_i)| + |prop(s_j)|} & if \quad prop(s_i) \cap prop(s_j) \neq \emptyset \\ \infty & otherwise \end{cases}, \qquad (1)$$

where, for a given entity $e \in S$, $prop(e)$ defines a set of relevant properties of e, expressed as follows.

- If e is a subprogram (procedure or function) then $prop(e)$ consists of: the subprogram itself, the source file or module where e is defined, the parameters types of e, the return type of e if it is a function and all subprograms that invoke e.
- If e is global variable then $prop(e)$ consists of: the variable itself, the source files or modules where the variable is defined, all subprograms that use e.

- If e is a user defined type then $prop(e)$ consists of: the type itself, all subprograms that use e, all subprograms that have a parameter of type e and all functions that have e as returned type.

We have chosen the distance between two entities as expressed in Equation (1) because it emphasizes the idea of cohesion. As illustrated in [14], "*Cohesion refers to the degree to which module components belong together*". Our distance, as defined in Equation (1), highlights the concept of cohesion. It is very likely that entities with low distances will be placed in the same application class, and distant entities will belong to different application classes.

For example, if a procedure p_1 has a formal parameter of type $\mathcal{T}$, and a procedure p_2 has also a formal parameter of type $\mathcal{T}$, intuitively the two procedures have to be placed into an application class corresponding to type $\mathcal{T}$. The attributes of the resulted application class will be obtained from type $\mathcal{T}$ and the two procedures p_1 and p_2 will be transformed in methods of the application class. As defined in Equation (1), $d(p_1, \mathcal{T})$ and $d(p_2, \mathcal{T})$ are small enough to favorize the grouping of procedures p_1, p_2 and type $\mathcal{T}$ into the same cluster, which confirms the above observation.

Based on the definition of distance d (Equation (1)) it can be easily proved that d is a semi-metric function, so a clustering based approach can be applied.

HOOS is based on the idea of hierarchical agglomerative clustering, and uses an heuristic for merging two clusters. We use *average link* as linkage metric, consequently we will consider the distance $dist(k, k')$ between two clusters $k \in \mathcal{K}$ and $k' \in \mathcal{K}$ as given in Equation (2).

$$dist(k, k') = \frac{1}{|k| \cdot |k'|} \cdot \sum_{e \in k, e' \in k'} d(e, e') \tag{2}$$

The heuristic used in *HOOS* is that, at a given step, the most two similar clusters (the pair of clusters that have the smallest distance between them) are merged only if the distance between them is less or equal to a given threshold, $distMin$. This means that the entities from the two clusters are close enough in order to be placed in the same cluster (application class).

The main steps of *HOOS* algorithm are:

- Each entity from the software system is put in its own cluster (singleton).
- The following steps are repeated until the partition of methods remains unchanged (no more clusters can be selected for merging):

– select the two most similar clusters from the current partition, i.e, the pair of clusters that minimize the distance from Equation (2). Let us denote by *dmin* the distance between the most similar clusters K_i and K_j;

– if $dmin \leq distMin$ (the given threshold), then clusters K_i and K_j will be merged, otherwise the partition remains unchanged.

We give next *HOOS* algorithm.

```
Algorithm HOOS is
Input: - the software system S = {s_1, ..., s_n}, n ≥ 2,
       - the semi-metric d between entities,
       - distMin > 0 the threshold for merging the clusters.
Output: - the partition K = {K_1, K_2, ..., K_p}, the OO structure of S.
Begin
   For i ← 1 to n do
      K_i ← {s_i} //each entity is put in its own cluster
   endfor
   K ← {K_1, ..., K_n} //the initial partition
   change ← true
   While change do //while K changes
      dmin ← dist(K_1, K_2) //the minimum distance between clusters
      i ← 1; j ← 1
      For i* ← 1 to n-1 do //the most similar clusters are chosen
        For j* ← i* + 1 to n do
          d ← dist(K_i*, K_j*)
          If d < dmin then
             dmin ← d; i ← i*; j ← j*
          endif
        endfor
      endfor
      If dmin ≤ distMin then
        K_new ← K_i ∪ K_j; K ← (K \ {K_i, K_j}) ∪ {K_new}
      else
        change ← false //the partition remains unchanged
      endif
   endwhile
End.
```

In our approach we have chosen the value 1 for the threshold $distMin$, because distances greater than 1 are obtained only for unrelated entities (Equation (1)).

Each cluster from the resulted partition will represent an application class from the equivalent object-oriented version of the software system S.

5. Example

In order to experimentally evaluate our algorithm proposed in Section 4, a simple code example is considered. The example is written in Borland Pascal, a procedural programming language developed by Niklaus Wirth as a small and efficient language intended to encourage good programming practices using structured programming and data structuring [15].

Let us consider a simple program that manipulates *lists* and *rational numbers*. We give bellow a code fragment from the Pascal source code that will provide the reader with an easy to follow example for transforming procedural code into object-oriented code using $HOOS$ clustering algorithm.

```
{...}
type List = record
                    s:array[1..100] of integer;   {the array of elements}
                    n:integer; {the size of the list}
            end;
type   Rational = record
                    numerator, denominator: integer;
                end;
{...}

procedure add(var l:List; poz:integer; e:integer);
{adds an element to a given position into a list}
 begin
  {..}
 end;

function size(l:List):integer;
{the size of a list}
 begin
  {..}
 end;

function element(l:List; poz:integer):integer;
{the element from a list from a given position}
```

```
 begin
  {..}
 end;

{...}
{other functions/procedures for lists manipulation}

procedure createRational(var r:Rational; n, d:integer);
{creates a rational number with a given numerator and a given denominator}
 begin
  {..}
 end;

procedure sum(var r:Rational; r1, r2:Rational);
{makes the sum of two given rational numbers}
 begin
  {..}
 end;

{...}
{other functions/procedures for rational numbers manipulation}
```

The first step of our approach is the **Data collection** step, during which the source code written in a procedural programming language is analyzed and relevant entities and the relations between them are extracted.

In order to extract methods, local and global variables, subprograms parameters, subprograms invocations and data types from a software written in Pascal programming language, ANTLR [17] parser generator framework is used. ANTLR is an open source Java tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. Using a grammatical description for the Pascal programming language, a lexical analyzer and a parser can be easily generated by ANTLR framework. Using the generated parser, the abstract syntax tree (AST) is constructed for the input source code and the needed entities from the Pascal source code are identified.

The set of entities extracted from the source code will contain all the subprograms (functions and procedures) defined in the software application and the defined data types from the system. This set will be used in the **Grouping** step as input for the *hierarchical* based clustering algorithm, *HOOS*. The idea is to group into separate clusters types and subprograms that are good

candidates for forming a class in the object-oriented version of the software system.

The relations between the extracted entities (subprograms invocations, type usages) are used for computing the distances between the entities as defined in Equation (1).

For the considered example, the size of the entity set is 15 and after the grouping step a partition with 2 clusters is obtained. The clusters corresponding to the presented source code extract are:

- Cluster 1:
  {**type List, add(List;integer;integer), size(List)**,
  element(List;integer):integer}
- Cluster 2:
  {**type Rational, createRational(Rational;integer;integer)**,
  add(Rational;Rational;Rational)}

Analyzing the obtained result, we can conclude that the obtained partition can be a good starting point for transforming a software system written in a procedural programming language into an object-oriented system. The clusters from the identified partition represent future application classes from the equivalent object-oriented version of the software system.

6. Comparison with Existing Approaches

In this section we will compare the approach presented in this paper with some existing related approaches which were described in Section 3.

In comparison with the approach from [4], our approach is a clustering based approach that heuristically determines the number of application classes candidates. In case of large software systems, this heuristic choice is an advantage, because it reduces the computational complexity of the transformation process.

The approaches from [1, 5] describe re-engineering processes for transforming procedural systems into object-oriented ones, but without focusing on automating the transformation process. Compared with these approaches, the approach presented in this paper offers an automatic method for the analyzed transformation.

The transformation processes presented in [16, 7] are non-clustering approaches, in comparison with our approach.

Our work differs from the approaches from [8, 9, 10, 11, 12] as the clustering based transformation process proposed in this paper exhibits a high level of automation.

7. Conclusions and Further Work

We have presented in this paper a hierarchical agglomerative clustering algorithm (*HOOS*) that can be used for assisting software developers in transforming procedural software systems into object-oriented ones.

We have also illustrated how our approach is applied for transforming a simple program written in Pascal into an equivalent object-oriented system. Advantages of our approach in comparison with existing similar approaches are also emphasized.

Further work will be done in the following directions:

- To improve the *distance* function used in the clustering process.
- To extend our approach in order to also determine relationships (class hierarchies) between the application classes obtained in the object-oriented system.
- To apply *HOOS* algorithm on large software systems.
- To apply other search based approaches for transforming procedural software systems into object-oriented ones.
- To use other unsupervised learning techniques (self organizing maps [18], Hebbian learning [19]) for transforming procedural systems into object-oriented ones.

Acknowledgement

This work was supported by CNCSIS-UEFISCSU, project number PNII-IDEI 2286/2008.

References

[1] Cobo, H., Mauco, V., Romero, M.d.C., Rodríguez, C.: A tool to reengineer legacy systems to object-oriented systems. In: ER '99: Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling, London, UK, Springer-Verlag (1999) 186–197

[2] Han, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2005)

[3] Czibula, I.: A clustering approach for transforming procedural into object-oriented software systems. In: KEPT '09: Proceedings of the Knowledge Engineering: Principles and Techniques Conference. (2009) 185–188

[4] Zou, Y., Kontogiannis, K.: Incremental transformation of procedural systems to object oriented platforms. In: COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications, Washington, DC, USA, IEEE Computer Society (2003) 290

[5] Jacobson, I., Lindström, F.: Reengineering of old systems to an object-oriented architecture. SIGPLAN Not. **26** (1991) 340–350

[6] Newcomb, P., Kotik, G.: Reengineering procedural into object-oriented systems. In: WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (1995) 237

[7] Gall, H., Klösch, R.: Program transformation to enhance the reuse potential of procedural software. In: SAC '94: Proceedings of the 1994 ACM symposium on Applied computing, New York, NY, USA, ACM (1994) 99–104

[8] Jacobson, I., Lindström, F.: Reengineering of old systems to an object-oriented architecture. SIGPLAN Not. **26** (1991) 340–350

[9] Jr., W.C.D., Nackman, L.R., Gracer, F.: Saving a legacy with objects. In: OOPSLA. (1989) 77–83

[10] Hausler, P.A., Pleszkoch, M.G., Linger, R.C., Hevner, A.R.: Using function abstraction to understand program behavior. IEEE Softw. **7** (1990) 55–63

[11] Gall, H., Klosch, R.: Finding objects in procedural programs: an alternative approach. In: WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (1995) 208

[12] Yeh, A.S., Harris, D.R., Reubenstein, H.B.: Recovering abstract data types and object instances from a conventional procedural language. In: WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (1995) 227

[13] Simon, F., Loffler, S., Lewerentz, C.: Distance based cohesion measuring. In: Proceedings of the 2nd European Software Measurement Conference (FESMA), Technologisch Instituut Amsterdam (1999)

[14] Bieman, J.M., Kang, B.K.: Measuring design-level cohesion. Software Engineering **24** (1998) 111–124

[15] Jensen, K., Wirth, N., Mickel, A.B., Miner, J.F.: Pascal User Manual and Report: ISO Pascal Standard. Springer (1991)

[16] Newcomb, P., Kotik, G.: Reengineering procedural into object-oriented systems. Proceedings of WCRE '95, USA, IEEE Comp. Soc. (1995) 237

[17] Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Raleigh (2007)

[18] Kohonen, T.: The self-organizing map. Neurocomputing **21** (1998) 1–6

[19] O'Reilly, R.C.: Generalization in interactive networks: The benefits of inhibitory competition and hebbian learning. Neural Computation **13** (2001) 1199–1241

Istvan Gergely Czibula and Gabriela Czibula
Department of Compuer Science
Babeş-Bolyai University
1, M. Kogalniceanu Street, Cluj-Napoca, Romania
email:{*istvanc, gabis*}*@cs.ubbcluj.ro*