Reverse Engineering Model from Object-Oriented Programs Using Concept Lattice

Hamed J. Fawareh
Faculty of Information Technology, Zarqa University, Jordan
fawareh@zu.edu.jo

Abstract

A reverse engineering model for object oriented programs using concept lattice is described in this paper. This model is based on applying the concept lattice for object-oriented features. These features are represented using concept lattice in the form of class and embedded representation. Te idea proposed here forms an analysis technique for object-oriented features based on representing the relations between various programs elements in a lattice structure. Furthermore, this paper demonstrates that a concept lattice and an embedded representation can facilitate the reverse-engineering of a class for which the source file is not available. It also, discusses how the lattice and the embedded method representation can be used in order to efficiently read source files if available.

Keywords: *Concept lattice, Reverse engineering, Object – Oriented program analysis.*

1. Introduction

Knowing how different entities are related leads to understanding a software application. In object-oriented application framework, entities are classes and methods. When one defines a class in an application, he requires knowledge about how behavior and structure have to be used using inheritance techniques. It is not trivial to achieve optimal use, especially when the number of classes is large or the inheritance hierarchy is deep [1]. In these situations, concept lattice can be used as a technique to help us cope with these problems, by visualizing the inheritance and interface relationships among the classes in the class hierarchy. Then the way, inheritance is used in the framework can be understood and documented, and this information is used to provide guidelines for how the framework can be modified or customized without running in to behavioral problems or without breaching the design conventions used when building the framework. This concept allows us to identify meaningful grouping of elements(objects) with common properties (attributes).

In this paper we provide a reverse engineering model, which represent object oriented features using concept lattice representation. This model provides information with the objective to support object-oriented program tasks, in addition to object oriented component relationships. All these views can help in understanding the source code of object oriented technique.

There are two aspects contributing to the complex nature of software system, which are behavior and structure. The response of a system to some input is referred to as the behavior of the system. To understand the behavior of software system means to understand the behavior of the system components and the relationships between these components. In object

oriented programs the features between various components of the software are the most important to understand the behavior of the software system.

Concept lattice analysis provides a way to identify groupings of objects that have common attributes. The mathematical foundation proved that for every binary relation between certain objects and attributes, a lattice can be constructed, that allows remarkable insight into the structure of the original relation [1].

The lattice concept started to be widely used in software technology [2-5], especially in applied concept analysis to software design, object oriented techniques and databases. The most related works to the presented approach in this work are applications of concept analysis to object oriented techniques[10-16]. Godin and Mili [7] used concept analysis to maintain, understand and detect inconsistencies in the Smalltalk Collection hierarchy. They automatically attempt to build a better interface hierarchy for class hierarchy based on convenience interface. In [9] lattice concepts are exploited to organize the set of classes into structured Galois lattice. Snelting [9] developed a tool based on computing concept lattice and displays features between configuration threads and visualizes the overall configuration structures. Snelting and Tip [1, 10] analyzed a class hierarchy in C++ and Java by making the relationship between methods and variables explicit. They were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. This approach proved useful to serve as a basis for automated or interactive restructuring tools for class hierarchies. Concept analysis is exploited in [1] for reengineering class hierarchies. A context describing the usage of class hierarchy is the starting point for the construction of a concept lattice from which redesign hints can be derived.

Siff and Reps used concept analysis to modularize legacy C programs into C++ classes [4]. Concept analysis is used to identify modules by considering both positive and negative information about the types of the function argument and the return values then construct a lattice concept from a program [4]. In [6] concept analysis is applied to extracting the code configurations while in [12] concept analysis is used to partitions the method of the class according to their use of fields and then presents them in concepts lattice form.

All the above approaches extracted the information and relationships for the implemented classes. More information and relationships (e.g., affect relationship) was not considered. This paper shows more relationships between classes which are analyzed in order to understand the kind of relationships that appear between object oriented techniques. It gives a different lattice representation for a class level, too.

Moreover, the concept of mathematical foundation to identify grouping of objects that have common features is adopted [19]. It has proved useful for understanding object-oriented programs. An elementary notation convention for object-oriented programs is discussed first.

2. Reverse Engineering

Reverse engineering is a process that is used in many daily applications we may frequently use in our lives. Reverse engineering can be defined as the analysis of a subject system process in which system components and their relationships are identified, in addition to creating representations of the system at a higher level of abstraction [7]. In other words, reverse engineering is a process of extracting information from a source code concerning software product design. The aim of reverse engineering is to remove ambiguity in the software and understand the software system with respect to facilities, enhancements,

redesign, and correctness. Moreover, reverse engineering provides facilities to help control many managerial problems [8-10].

The tasks involved in reverse engineering are analysis of a subject system process, which are identified systems components and their interrelationships [11]. The maintainer spends a lot of time to understand all the activities of the problem and to correct it. Therefore the focus of the reverse engineering process is to aid program understanding. In order to meet this object in the above framework the reverse engineering effort must address the two central issues in reverse engineering, namely, knowledge representation and automated extraction of the knowledge representation [7][13]. The first object knowledge representation explicitly represents the comprehensive activities of a programmer [12]. The object of second issue, knowledge representation model is to ensure that the representation model as part of reverse engineering approach is automatically extractable from the source code [18].

3. Lattice Model

The original concept of Lattice is introduced by Birkhoff [4]. It refers to a technique to identify grouping of objects that have common attributes, which is to allow remarkable insight into the structure of original relation[17]. Ever since the introduction of this concept lattice, various slightly different notions for program dependence have been proposed, together with methods for computing them. To serve our purpose, we choose to extend the traditional concept of mathematical Lattice to cater for program relations that are more amenable to object-oriented programs. Lattice technique for object-oriented programs would involve capturing various combinations of features between classes as well as their components that are considered important in software system. This section discusses definitions of several concepts for lattice technique that bear strong relationships with features of interest.

Definition:

An elementary context is a 3-tuple $C=< E_1, E_2, R >$ where,

1. *Ent(P)* denotes the set an identifier (name) of a class or method, or a labelling of any statement or variable in *P*.
2. Relationships $R \in Dep(P)$,
3. *Mtd(A)* denotes the set method declared in the fields of class *A*.
4. *Var(P)* denotes the set variables in class *A* .
5. $E_1, E_2 \in Ent(P)$ and the expression $E_1 R E_2$ is valid.

The context *C* can be used to identify the features of interest that may exist between entities E_1 and E_2 in *P*.

Features can also be categorized according to levels. The first category, *class-level* involves features of a class to another class. The second category, *method-level* involves features of a method or a statement within a method to another method or variable in a class. The final category, *statement-level* which are basically intra-method involving statements within a given method. Context, in turn, can also be categorized according to such levels of features as they intend to identify object.

The most general type of contexts is the *class-level*, which identifies *class-level* features. This type of context intuitively excludes less number of elements compared to the lower level of similar form of criterion. The general form of context criterion is $C = < E1, E_2, R >$, where $R \in \{ i, ni, u, nu, a, na \}$.

The concept analysis shows how any variable in *V* is being used or affected by class *A* through class *B*. Essentially it will identify class *A*, *B* and all the intermediate classes linking *A* to *B* if *A* and *B* are related through such features. However, if this is true, the parts of *B* and intermediate classes that are related to variables in *V* with respect to the dependence relation of interest are identified only. Method-level gives more refined view of the use and affects features between classes.

- **Lattice Layout**

In this section, we provide a description of feature graphs for representing meaningful features between entities of object-oriented programs. A formal description of the features relations of interest is given before giving a representative illustration of object-oriented program feature graphs using concept of lattice. A lattice representation approach represents software components that will be affected due to any particular modification being made to a certain component. The paper also discusses an approach for understanding object-oriented programs through the use of lattice representation approach. The approach proposes the construction of an automated tool to extract the feature information from the source code. This model should be used interactively in the software phases to locate the features of a given component of an object-oriented program [10].

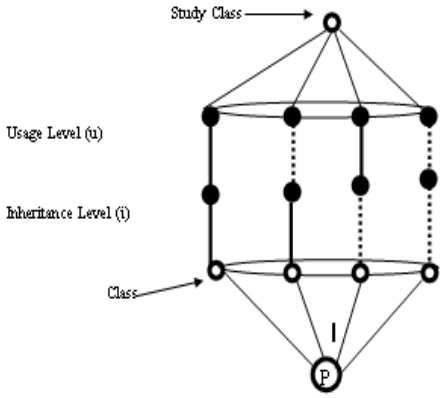

Figure 1: Cylindrical Lattice Representation

- **Cylindrical Lattice Representation**

In the cylindrical lattice representation, we represent the lattice relationships between object oriented techniques in simple way. To explain this representation we start with the class C as a study class in a program P. We represent the relationships as edge between class C and other classes in the same program. The lower base represents the program P and the upper represent the class C. The cylinder may have several levels, each level represents an attribute. Figure 1 shows the layout for cylindrical lattice representations with two levels only, namely i and u.

- **Package Lattice Representation**

The cylindrical lattice representation given in the previous section is suitable for program with several classes. However, package lattice representation is suitable for several programs can be considered. This lattice representation can be generated from cylindrical representation of single programs connected at joining points as related in the package. The

diagram may get extremely complicated with huge number of classes and relationships, however the package lattice representation simplifies the software system relation diagram remarkably this can achieved in several different layouts. Examples of the package lattice representations are shown in figure 2. They represent two cases, for simplicity, suppose class C have relationships with other classes in programs P1, P2 , P3 and P4. The first case, nested lattice representations will appear as shown in figure 2-A. This representation shows all relationships between class C and all programs and classes at the same software package.

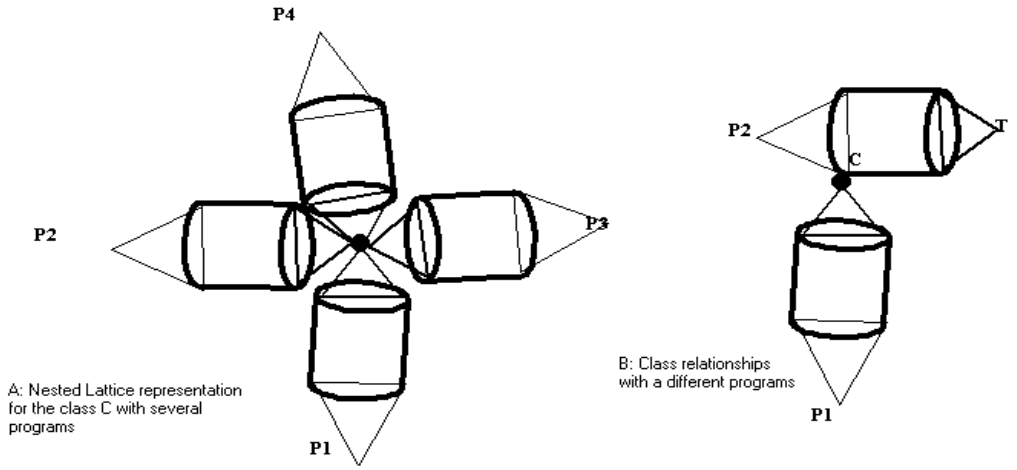

Figure 2: Lattice Package Layout

The second case, suppose class C in a program P1 is related to class T in a program P2. The lattice representation in figure 2-B shows the nested representation between programs P1 and P2, by selecting class C and class T to represent the study classes. This lattice representation also shows the relationship between the two classes with other software system.

4. Lattice Graph Representation for Class Level

In the section we demonstrate the use of concept analysis on a class level. The system automatically extracts the information about the classes and accesses data. Concept analysis is performed on a three main relationships between classes that can be emphasised and need to be represented on a binary relation based on the following roles:

1. Inheritance: $A \xrightarrow{i} B$ if and only if A is a subclass (derived class) of B.
2. Class using class: $A \xrightarrow{u} B$ if and only if either one of the following holds:
   a. $\exists\, m \in Mtd(A)$ *and* $v \in Var(B)$ such that $A.m \xrightarrow{u} B.v$.
   b. $\exists\, m \in Mtd(A), n \in Mtd(B)$, such that $A.m \xrightarrow{u} B.n$.
2. Class affecting class: $A \xrightarrow{a} B$ if and only if $\exists\ m \in Mtd(A)$ and $v \in Var(B)$ such that $A.m \xrightarrow{a} B.v$.

The proposed lattice concept can be applied within software system tasks particularly for understanding the features among elements of object-oriented programs. We foresee that normal approach would be to formulate meaningful table relating various aspects of interests see table 1. This table can be analysed and utilised. It includes all possible objects features to a single object (PartNumber in this case).

Figure 3 presents a small example of five classes, the goal was to show how lattice concept analysis can be used in order to understand this point of view. An analysis of Java code program is conducted as an example for the lattice concept in order to clarify the features. Hence, the context definition for this concept analysis starts with a triple $C = < E_1, E_2, R>$ for class *PartNumber* proposed as:

$$C = <PartNumber, */class, R>$$

Where; */class represent the objects E_2 that is restricted to the element "class" only, as this example concentrates only on class relationships, R represent the relationships among classes. Only three relations were considered here, namely inheritance, usage and affect.

Table 1 describes the features between class *PartNumber* and all other classes in program P. Figure 3-A shows class *PartNumber* relations for this program that contains five classes having several class features. Figure 3-A shows also a lot of interferences between horizontal nodes; this modular structure is not good and even hard to understand, particularly for a huge software system. These interferences can be detected and removed by algorithm transformations using lattice concepts, figure 4. Figure 3-B and 3-C display the corresponding lattice, horizontally decomposed. It is connected only via bottom up element. The transformation algorithm, of figure 4, represents an algorithm for transformation from complicated graph of figure 3-A to the lattice diagram of figure 3-B. For example, the relation between class *PartNumber* and *PartAssembly* denoted by $PartNumber \xrightarrow{R} PartAssembly$ is read from figure 3-A as *{na, nu, i}* by following three paths, while in figures 3-B and 3-**C,** it is simply read from the single path connecting *PartAssembly* with *PartNumber*.
This shows how important the lattice concepts in reducing the relation difficulty. Furthermore, the relations between all classes in P with *PartNumber* are denoted by $PartNumber \xrightarrow{R} *$ (where * represent all classes in P) is clearly shown in the lattice matrix and far more easily readable.

```
Public class PartNumber
public class Part
{
public   double Cost(){
  const = 0;
  private:
    PartNumber itsPartNumber=  new PartNuber();
   String Description;
     }
}
public class Assembly extend Part
{
  Public double Cost(){…}
  Part itsParts = new Part();
}
```

```
class PiecePart extend  Part
{
     public  double Cost() {
   private double itsCost;
        PartNumber itsPartNumber = new
PartNumber();
}
}
Public  class PartAssembly extend Part
{
private PartNumber
   Assembly aAssembly= new Assembly();
 cost(){…..}
}
```

Figure 3: Simple Program

Table1 . class relations for class *PartNumber*

	PartNumber					
Attributes	i	ni	u	nu	a	na
Part		x	x		x	
Assembly		x		x	x	
PiecePart		x	x		x	
PartAssembly	x			x		x

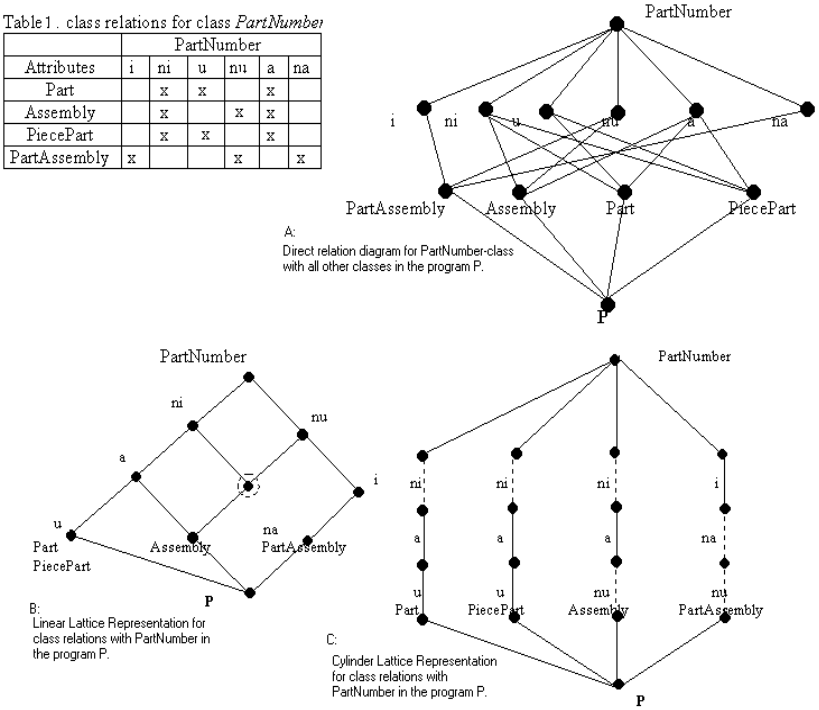

Figure 4: Lattice Representation

Input: a program P, class A.
Output: linear lattice representation

Method
Begin
Select a class A from a program P
Search for all features relations
Search all classes has unique relationships Between where d unique with y
Draw a node p.
Find a lattice dummy node
Repeat
Find a group of classes that have a common relation ships with class A.
Start with the first node in lattice diagram that has a common relationship with A.
Find a path from $Y_i \overset{d}{\Rightarrow}$ A through the relation $Yi \overset{d}{\Rightarrow}$ A where d={i, u, a, ni, nu, na}
Until no more classes Y_i = {}.
End

Figure 4: Lattice Algorithm

Furthermore, we describe a prototype implementation for lattice concept principle in order to visualize the lattice cylinder representations. Another java code program is used as an example for this implementation.

5. The embedded method representation (Method level)

In the section we demonstrate the use of concept analysis on a Draw3D class. The system automatically extracts the information about the methods and accesses data. Concept analysis is performed on a binary relation between set of behaviors and attributes based on object oriented program features.

In the analysis of Draw3Dclass, we use the attributes and the class behaviors as features. Hence, we use the binary relation to specify the class behaviors used or affect each attributes. The system is classified into direct or indirect embedded method representations.

We obtain the embedded method representation which provides a more detailed visualization of the class. Since a concept lattice for methods use the same set of fields, it does not provide information about the interaction between these methods. Examining the concept lattice does not reveal whether a method accesses a combination of fields directly, by accessing their values, or indirectly, by invoking methods that access them directly. By superimposing the method representation on the class concept lattice,

A method representation is a graph derived from the cylinder representation in which nodes represent methods and edges represent method-invocations. This graph is a common means of visualizing the interaction between the methods of a class.

In the embedded representation, the methods of each concept are grouped together. Groups are explicitly marked and are connected with edges, creating the lattice structure. In order to demonstrate how the embedded representation can provide important information which does not appear in the concept lattice, suppose that the class contains method, named setYX, which modifies the values of the x and y fields. If method setYX modifies the two fields by invoking methods setX and setY directly, then the corresponding embedded method representation shows the symmetry between methods setYX and setXY.

Unlike general graph layout algorithms, an embedded method representation layout is based on semantics, and group related methods together. Also, we believe that because of its inherent properties, an embedded method representation layout has less crossing edges than an unoptimized graph layout. We base this claim on the property that an edge leaving a method in some concept in the embedded method representation can only reach a method in the same concept or in another concept that is dominated by the first. Therefore, the edges for method calls that occur in separate parts of the class do not cross in the embedded method representation.

6. Implementation and Result

We have implemented a model prototype that employs lattice concept to achieve visualization suitable for understanding object oriented program. This prototype takes an object oriented program and builds a context based on object oriented features. The system builds lattice cylindrical representation and investigates on a class level. In particular, we have used the prototype model for testing various programs and systems written in java. Figure 5 presents an example of eight classes having various relationships which are organized in several common blocks. After the features table is built for a selected class of them, for example the class Geometric Object, the features graph is constructed as in figure 5-A. It shows so many features, making it difficult to follow. Applying the lattice concept for such result, would lead to the modularized lattice diagram in cylindrical representation, figure 5-B. The model allow any class to be chosen in order to see its relationship with all others classes as shown in figure 5-D and 5-C.

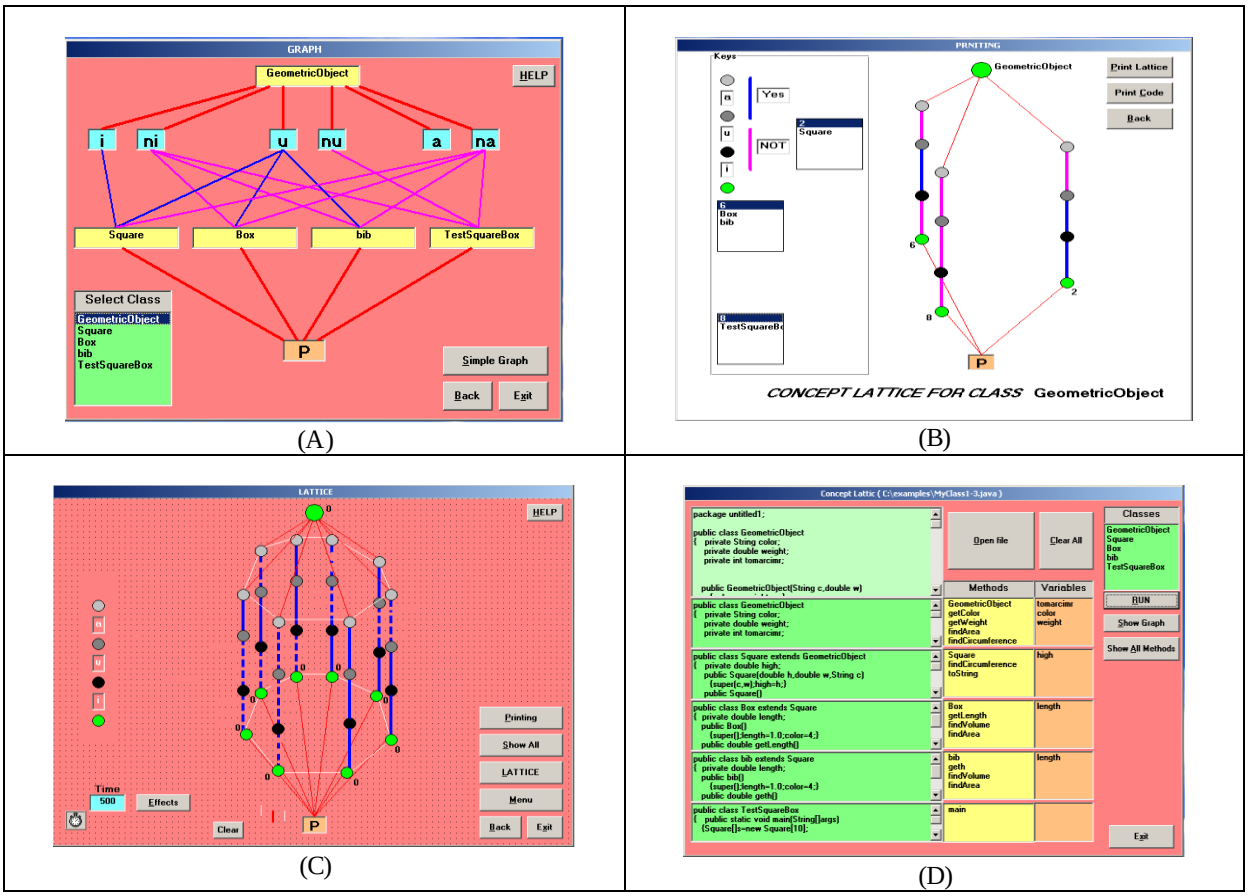

Figure 5: Prototype and System Result

7. Conclusion

This paper provides the overall architecture of the proposed lattice model. The model is embedded in the software engineering environment. We also, discusses concepts lattice suitable for understanding object-oriented programs. We provide some insights into the relation behind the high-level relationships between the components. Class and component features are more natural in supporting the process of understanding the structure and behaviour of object oriented programs. Based on the identified grouping of objects that have common features, concepts lattice have been developed for understanding object-oriented programs. We also highlight the potential application of this new approach by providing an illustration for cylindrical lattice representations. This paper discusses an embedded method representation which can be used in order to efficiently read the source file.

References

[1]. Snelting, G. and Tip, F., "Reengineering Class Hierarchies Using Concept Analysis", ACM Trans. Programming Languages and Systems, 1998.

[2]. D. G. Kourie, S. Obiedkov, B. W. Watson, D. V. D. Merwe, An incremental algorithm to construct a lattice of set intersections, Science of Computer Programming, 74(2009), 128-142.

[3]. J. Muangprathub, V. Boonjing and P. Pattaraintakorn, A new case-based classification using incremental concept lattice knowledge, Data & Knowledge Engineering, 83(1) (2013), 39-53.

[4]. Jirapond Muangprathub, "A Novel Algorithm for Building Concept Lattice" Applied Mathematical Sciences, Vol. 8, 2014, no. 11, 507 – 515.

[5]. Ra'Fat Al-Msie, Marianne Huchard, Abdelhak Seriai, Christelle Urtado and Sylvain Vauttier "Reverse Engineering Feature Models from Software Conjurations using Formal Concept Analysis" CLA 2014: Proceedings of the Eleventh International Conference on Concept Lattices and Their Applications pp. 95-102, 2014

[6]. Krone, and Snelting, "On the Inference of Configuration Structures from Source Code", Proc. 16th International Conference on Software Engineering, May 1994, IEEE Comp.Soc. Press, PP. 49-57.

[7]. Linding, C. and Snelting, G., "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis", Proc. 19th International Conference on Software Engineering, May 1997, IEEE Comp. Soc. Press, PP. 349-359.

[8]. Siff, M and Reps, T., "Identifying Modules via Concept Analysis", Proc. International Conference on Software Maintenance, Bari 1997, PP. 170-179.

[9]. Snelting, G. "Software Reengineering Based on Concept Lattices." In Proceeding 4th European Conference on Software maintenance and Reengineering, page 3-12. IEEE, 2000.

[10]. Snelting G. "Reengineering of Configurations Based on Mathematical Concept Analysis", *ACM Trans. On Software Engineering and Methodology, vol. 5, no. 2, pp. 46-89, 1996*.

[11]. Godin R. and Mili H. "Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices", In proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications, pages 394-410. CM Press, 1993.

[12]. Birkhpff G., "Lattice Theory", America Mathematical Society, Providence, R. I. 1st edition 1940.

[13]. Godin R., Mili H., Mineau G. W., Missaoui R., Arfi ., and Chau T., "Design of class hierarchies based on concept (galois) lattices". *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[14]. Snelting, G. and Tip, F., "Understanding Class Hierarchies Using Concept Analysis", ACM Trans. On Programming Languages and Systems, pages 540-582, May 2000.

[15]. Funk P., Lewien, and G. Snelting, "algorithms for concept Lattice Decomposition and their application", Technical Report, Computer science Dept, University of Technische, Braunschweig, 1995

[16]. Dekel, Uri; Gil, Y. "Revealing class structure with concept lattices", Proceedings of WCRE: 10th Working Conference on Reverse Engineering, 13-16 Nov. 2003. pp. 353 – 363, DOI. 10.1109/WCRE.2003.1287267.

[17]. Duquenne, V., Chabert C., Cherfouh A. and Doyen A., "Structuration of Phenotypes and Genotypes through Galois Lattices and Implications", Applied Artificial Intelligence 17:243-256. 2003.

[18]. Chaudron L., Maille N., and Boyer M.., "The Cube Lattice Model and Its Applications", Applied Artificial Intelligence 17:207-242. 2003.

[19]. Ahmed Fouad Ali, "Accelerated Bat Algorithm for Solving Integer Programming Problems", Egyptian of Computer Science Journal, Volume 39, Issues No. 1, 2015.