A Bio-Inspired Approach to Selective Inheritance Modeling

Darin Hamouda, Said Ghoul, and Hanan Hardan

Faculty of Information Technology, Philadelphia University, Amman, Jordan
{dhamoudeh, sghoul, hhardan}@philadelphia.edu.jo

Abstract

The conventional inheritance concept adopted in the current Object Oriented Programming (OOP) was applied more than three decades ago. It is acting on “is-a” hierarchy model, and has some defects; while OOP is trying to be more close to the real life, it is still far from genetics principles. Adopted inheritance concept means that the child class can inherit, and get everything that is public in the parent class automatically. This process has solved many problems, but it does not simulate what is really happening in our life, where each object can gain just the needed properties from the parent class. So, in simple words, conventional inheritance is not selective and is generating identical objects. One of introduced solutions was the selective inheritance. But, while conventional inheritance acts on “is-a” hierarchy model, the works that introduced the selective inheritance were also done on this model and still suffers from some problems. The inspiration from the “real life” genetics has led us to a selective inheritance acting upon a “Composed by” model rather than the “is-a” model, where the properties are classified into several classes according to "Composed by" relation. This paper proposes a Genetic approach to a new model for selective inheritance that is replacing the “is-a” hierarchy model by a “Composed by” one, which reduces the complexity and makes each object has its own definition. After that, it proposes a language extension to be adopted in any object oriented programming language.

Keywords: Bio-inspired systems modeling, Is-a hierarchy model, composed-by hierarchy model, selective inheritance

1. Introduction

Bio-inspired approach became more widespread nowadays to introduce natural solutions for artificial systems problems. This concept is one of several approaches classified among the Natural Computing approaches. It has been used in many areas and to develop problem-solving techniques such as, Evolutionary Computing, Neuro-computing (neural networks), Immuno-computing, Swarm Intelligence, and also in Software Modeling [1, 2].

Since Object Oriented Programming has been created, in which everything in a program is an object, which tries to mimic real life, a lot of concepts have been adopted, and one of the most important is the inheritance. Inheritance means that the child class can inherit, and get everything that is public in the parent class automatically [3, 4]. Although this process minimizes the amount of duplicate code in an application by putting common code in a parent class and sharing it amongst several child classes, it will produce identical objects for the same class. And as known, inheritance is applied on a “is-a” hierarchy model, and while slight properties difference needs new class creation this will lead to a complex inheritance hierarchy.

Several researches have been done to introduce proposals to these problems. The Selective Inheritance (S.I) was the promising approach [5-7]; where through it we can limit the amount
of information in each object, by inheriting the needed properties from specific ancestors, and solve overlapping problems, by choosing the appropriate property from two or more contradictories.

Although several works have agreed on the importance of using selective inheritance concept [5, 8], and others have proposed it as a solution to use in their works [6, 9, 10], the proposed S.I approach did not completely solve all conventional inheritance problems; because it also has been applied on the “is-a” model, and even worse, it has created two other new problems: where should the property be chosen, at class level or at object level? And it was not supported by any object oriented programming language. From all of that, and based on other studies, and believing in inspiring from nature, we propose a Genetic approach to a new model for selective inheritance that is replacing the “is-a” hierarchy model by a “Composed by” one, which reduces the complexity and makes each object have its own definition. After that we propose a language extension to be adopted in any object oriented programming language. In the following we survey some similar significant approaches to selective inheritance in order to identify the insufficiencies justifying our approach, then the genetic approach to inheritance modeling will be introduced, and finally its evaluation will be outlined.

2. Similar Works on Selective Inheritance Approaches

The selective inheritance, in actual approaches, is mainly provided at two exclusive levels: class level and object level.

2.1. Selective Inheritance at Class Level

In approach presented in [11], the parent (class) holds two types of parameters: inheritable which is referred to as “gene” fields and non-inheritable which is referred to as “non-gene” fields. All the inheritable parameters will be inherited into the child (object). Therefore, the object does not have the ability to select only the parameters that it needs.

Another selection at this level was proposed in [12] to avoid several problems that may arise when applying conventional inheritance such as inheriting contradictory or repeated features, complex objects when inheriting all features from all ancestors and other problems. It is to allow the class to dynamically select any ancestor subset; this is done by using embryonic class notion, where the embryonic class contains a default attribute called ancestor-list, which is a list of ancestors from which the class inherits its features. An implicit method “Formulate” accepts the ancestor-list as a parameter and constructs the internal structure accordingly. With selective multiple inheritance, a class may inherit features from any number M of a given set of N ancestor classes. Many of this approach ideas were inspired from the inheritance in biological systems where the observed features do not expose the entire genetic information. Unfortunately this approach is only convenient for multi-level multiple inheritance, beside that the object is not the one who determines whom to inherit from and even the object will be forced to inherit all the features from the ancestors’ subset.

2.2. Selective Inheritance at Object Level

These approaches are the most logical and preferable approach to selective inheritance, where the object can easily select the needed features from any class(s) that holds those features. Therefore, the approaches took two trends: genetic and non-genetic approaches.

Non-Genetic Approaches, The selective inheritance concept was used in a different perspective [13], where, implementation solutions were proposed to derive a subclass with
reducing the redundancy in representation and reducing the number of methods redefinition. These solutions were by using suitable names for the class operations, using inherited names, based on pointer data members, or based on read and write operations. This approach did not meet the real objective of the selective inheritance which is the explicit and coherent selection or rejection of any property in the class hierarchy according to some rules.

**Genetic Approaches**, These approaches are inspired by biological systems especially the genetic process. The Work [14] has proposed a new approach to classification which they call semantic classification. By semantic classification, authors mean the possibility of making objects which have different profiles (i.e., properties) but identical underlying semantics, instances of the same class. The difference among objects will be achieved by choosing appropriate properties before creating those objects. Since a semantic class contains all possible properties of all varieties belonging to, a problem may arise if some of these properties are incompatible or exclusive. Thus, selecting the properties for an object is necessary and must take into account those incompatibilities. To deal with this situation, authors described the alternation of properties or classes. Properties alternation is a concept that deals with variety of properties, whereas classes’ alternation is an intermediate form of specialisation between simple specialisation and multiple one. *Alternation of properties* consists mainly of defining in the same class one or more properties in multiple versions. Alternatives of the same property are exclusive. A given object cannot possess more than one version. Thus a definition of properties that are different for objects of the same class is possible.

The determination of properties that an object of the class holds is based on the interpretation of a program associated with that object. This program is called genetic program. It is composed of a set of rules that reject or select properties from different classes of the Is-A hierarchy and resolving conflicts at the same time. This work had reduced the number of classes. But this approach did not propose a full bio-inspired model. The semantic classification was done on “is-a” hierarchy model and the genetic relations between features are not modeled.

Full bio-inspired approaches were done in [15, 7]; where, in the first work, several general principles to model the genetic selective inheritance were proposed. This work is a general platform for the object oriented genetic class modeling principles, allowing definition of coherent views. Objects are instantiations of those views.

### 2.3. The Presented Work

Based on the above presented researches, our work prolongs mainly those dealing with genetic selective inheritance approaches at the object level, especially [7, 15]. It mainly develops and formalizes two of the general principles presented in [15]: (1) each artificial entity (phenotype) is an instance developed from a view (genotype) on Genetic class (genome). And (2) the nature of a view is completely operative. The work [7] proposed a bio-inspired approach to support Aspect-Oriented paradigm. The author has used several genetics concepts including Genome (Genetic class), Genotype (view), and Phenotype (instance) to support the Aspect-Oriented design. According to that, the author proposed a general extension, to Object Oriented Programming Languages, allowing the definition of: (1) Different versions of attributes and methods in the same class, (2) Versions compatibility rules, and (3) configuration instantiation. In our work, these two principles were revised, enhanced, specified, and formalized.
3. A Genetic Approach to Inheritance Modeling

In our approach, to inheritance modeling, we aim to be as close as possible to the real world. For that, we were inspired by the genetics processes which are the basis of living organisms. This was to solve many problems that object oriented languages did not solve, or to modify some concepts to be more efficient in use. Selective inheritance, which is one of the inheritance properties, that is not supported by any of object oriented languages, is offset by the genetic genotyping process which is the main process in producing organisms with different characteristics in our real life. Both, selective inheritance and genotyping are the processes of selecting desired traits and functions from a set of all possible characteristics.

3.1. New Definitions

In the following, we will present some basic conventional Object Oriented (OO) concepts and their new redefinitions in our work: Class, attribute, method, composition, inheritance, and instance.

**Class, A Conventional OO Class** Figure 1 is a set of objects that share common attributes and methods. In other words, the class groups objects that have similar structural and behavioural properties. Objects that have slightly different profiles must belong to different classes. A Genetic Class is a semantic class Figure 2, introduced in [14], which groups objects having different structural and behavioural properties. But each object owned properties are subset of its class properties. So, a semantic class groups together objects holding subsets of its properties. In addition, a genetic class contains some new concepts as shown in Figure 3.

**Attribute, A Conventional OO Attribute** is a specification that defines a data structure of an object. Values of attributes form an object state. Each attribute is defined by a single data structure (i.e., Data-Type, atti :). A Genetic Attribute has a set of alternative definitions (i.e., Atti= alternatives {alt1, alt2... alti}). An object must hold coherent alternatives from each “needed” attribute.

**Method, A Conventional OO Method** is a subroutine (or function) associated with a class. In other words, it is an action which an object is able to perform. It is defined by a single way (i.e., methi () { // method body}). A Genetic Method is method having a set of alternative
definitions (I.e. $\text{Meth}_i() = \text{alternatives \{met}_1, \text{met}_2... \text{met}_n\} :$). An object must hold coherent alternatives from each “needed” method.

**Composition.** A Conventional OO Composition allows class to contain instance variables that refer to other objects (simple or composed). Each class instances holds these "referred to" objects. A Genetic Composition allows class definition to be composed of other genetic classes’ definitions. An instance of this class may hold only selected attributes and methods from selected "composed by" classes. So, it may not include a "composed by" class components at all, include some part of it, or include it completely (with one selected alternative for each component). The example Figure 4 illustrates this definition. The class “Transportation” definition is composed by the classes: aerial, maritime, and ground definition; which is different from the attribute composition concept in the conventional object oriented paradigm. Each instance may hold its properties selected from any one of these classes, i.e., the object $T_i$ defined as it follows: $\text{Transportation selection}_i$, $T_i$ Holds properties defined by the selection $\text{Selection}_i$.

**Inheritance.** The Conventional OO Inheritance is based on the “Is-a” relation where a subclass inherits all the attributes and methods of its parent class(s). The Genetic Inheritance is based on a selective inheritance program that selects only needed attributes/methods from a specific class and its "composed by" classes. This selection program replaces the "Is a" relation semantics Figure 5/a, which defines implicitly a total inheritance, whereas this program defines an explicit and restrictive inheritance Figure 5/b.

**Instance.** A Conventional OO Instance is a value of a class having its behaviours. According to that, all objects instantiated from a class are similar; where they will all have the same structure (attributes /methods). A Genetic Instance (O) may be created from a genetic class (C), according to an explicit selection program (P) that selects its needed attributes and methods alternatives from (C). According to that, objects created from the same class may differ from each other. Example:  

$\text{Transportation selection}_1$, $T_1$ // Assuming selection$_1$ selects aerial  
$\text{Transportation selection}_2$, $T_2$ // Assuming selection$_2$ selects maritime & aerial.

3.2. Genetic Class (Genome) Modeling

In the following we will develop a Genetic class (Genome) and a View (Genotype) models from the selective inheritance viewpoint.
The genetic patrimony, genome, of a species includes the definition of all its possible characteristics (organic, functional, and behavioral) along with the information controlling their coherence [15, 16]. Each characteristic might be developed in alternative ways. Each way constitutes a version of this characteristic. We model the selective inheritance viewpoint of a Genome by a genetic class containing a composed-by hierarchy of Genetic classes. The (Figure 6) completes the Figure 3 by introducing some structural details.

### 3.3. View (Genotype) and Instance (Phenotype) Modeling

Genotype [15, 16] is a genetic configurator generating a physical organism (phenotype) or a group of organisms from characteristics or traits versions. The chosen characteristics versions must be coherent. We model a Genotype of a genome by a View (selective inheritance program) on a Genetic class, Figure 6 and a phenotype of a genotype by an instance of a view Figure 7. To ensure and control the coherence of an instance, the following dependency relations between characteristics versions are managed:

- **AreExclusive**: These relations ensure the exclusion between characteristics versions. The view, being coherent by construction, holds normally non-exclusive characteristics versions. This is defined by the following rules:
  - Enable characteristic-version, Excludes Enable characteristic-version,
  - Disable characteristic-version, Excludes Disable characteristic-version,

- **AreImplied.** These relations ensure the implication between characteristics versions. This is supported by the following rules:
  - Enable characteristic-version, Implies Enable characteristic-version,
  - Disable characteristic-version, Disable characteristic-version,

### Figure 6. A Genetic Class Structure

```plaintext
Class Genetics-class-name
{Composed by Genetics-based classes list>
<Attributes alternative definition>
<Methods alternative definition>
Control Rules
<Definition of rules that govern the classification>
<Definition of rules that control the coherent selection of object properties from the alternatives>
// {AreRelatedToAspect, AreExclusive, AreImplied, AreDominant, AreView}.<Definition of rules that control the instances behaviors>
}
```

### Figure 7. Genetic Class (Software Database), View, and Instance

```plaintext
View view-name //on a genetic class

// we specify what to be enabled.
Enable <attributes versions, Functions versions>

// we specify what to be disabled.
```

### Figure 8. View Model – Selective Inheritance Program
The View model Figure 8 consists of:

**Enable:** Allows an instance of this view to hold a set of explicitly enumerated attributes and functions versions. This set is implicitly augmented by the dependency relations AreImplied.

**Disable:** Allows an instance to lose a set of explicitly enumerated attributes and functions versions. This set is implicitly augmented by the dependency relation AreImplied. The disabled properties are inactive for that instance.

### 3.4. View Model Interpretation – Selective Inheritance Process

At first, a Genetic class definition is introduced as a Genetic Class. Second, a set of wanted characteristics versions is defined to be enabled, or a set of unwanted characteristics versions is defined to be disabled. This is achieved by a View Program. Then, both the Genetic class and the View Program are set as inputs for the interpretation process, which is the selective inheritance process, to produce the desired View. It is worth mentioning that the previous interpretation process will be applied on the Genetic class which contains the full set of characteristics that are classified into several classes according to “composed by” relations and there will not be any inheritance relations between the inner classes. This classification model provides an easy, flexible and more logical selection of properties. Instead of inheriting from several super-classes and several ancestors according to the “is-a” hierarchy, in our approach, an object will select its needed properties from “composed by” classes. The selection process Figure 9 is acting in breadth; while searching for the appropriate and compatible class, and in depth while deepen inside specific class to search for wanted and compatible characteristics versions.

```plaintext
Process View-Interpretation (input Genetic class, View program) // Selective Inheritance process
{
    If Enable <> null then add Enabled-characteristics versions to EnabList
    If Disable <> null then add Disabled-characteristics to DisabList

    Select-coherent-characteristics versions (EnabList, DisabList);
    // will select (Enable) and output the compatible attributes, functions, and behavior versions
    // and produce the corresponding View.
}
// View-Interpretation
```

**Figure 9. Interpretation Process – Selective**

The interpretation process of a View definition Figure 9 is mainly supported by the dependency relations control which ensures the coherence of the generated view. It is enforced by the following rules:

- **Initial state: Elements and coherence**
  - **R1.** Let EnabAttr, EnabDef, EnabFun and EnabBeh be, respectively, the lists of the Attributes, Attributes-definitions, functions, and behaviors to be enabled.
    
    - EnabAttr ← Attributes (imposed in Enable clause);
    - EnabDef ← Attributes-definitions (imposed in Enable clause);
    - EnabFun ← Functions (imposed in Enable clause);
    - EnabBeh ← behaviors (imposed in Enable clause);
The coherence of EnabAttr, EnabDef, EnabFun, and EnabBeh is checked separately because there is no Exclude relation between Enabling Attributes, Enabling Attributes-definitions, enabling functions and enabling behaviors. This coherence deals especially with:

1. The existence of the Attributes, Attributes-definitions, functions and behaviors, and,
2. The verification that the elements of each list do not exclude elements of the same list.

R2. Let DisabAttr, DisabDef, DisabFun and DisabBeh be, respectively, the lists of Attributes, Attributes-definitions, functions and behaviors to be disabled.

\[
\begin{align*}
\text{DisabAttr} & \leftarrow \text{Attributes (discarded in Disable clause)}; \\
\text{DisabDef} & \leftarrow \text{Attribute-definitions (discarded in Disable clause)}; \\
\text{DisabFun} & \leftarrow \text{Functions (discarded in Disable clause)}; \\
\text{DisabBeh} & \leftarrow \text{behaviors (discarded in Disable clause)};
\end{align*}
\]

The coherence of DisabAttr, DisabDef, DisabFun and DisabBeh is checked separately because there is no Exclude relation between Disabling Attributes, Disabling Attributes-definitions, disabling functions and disabling behaviors. This coherence deals with the same conditions of R1.

R3. The coherence of EnabAttr with DisabAttr is checked through the predicate: \(\text{EnabAttr} \cap \text{DisabAttr} = \emptyset\). Each element of EnabAttr doesn't imply directly or indirectly an element of DisabAttr, and each element of DisabAttr doesn't imply directly or indirectly an element of EnabAttr.

The coherence of EnabDef with DisabDef is checked through the predicate: \(\text{EnabDef} \cap \text{DisabDef} = \emptyset\). Each element of EnabDef doesn't imply directly or indirectly an element of DisabDef, and each element of DisabDef doesn't imply directly or indirectly an element of EnabDef.

The coherence of EnabFun with DisabFun is checked through the predicate: \(\text{EnabFun} \cap \text{DisabFun} = \emptyset\). Each element of EnabFun doesn't imply directly or indirectly an element of DisabFun, and each element of DisabFun doesn't imply directly or indirectly an element of EnabFun.

The coherence of EnabBeh with DisabBeh is checked through the predicate: \(\text{EnabBeh} \cap \text{DisabBeh} = \emptyset\). Each element of EnabBeh doesn't imply directly or indirectly an element of DisabBeh, and each element of DisabBeh doesn't imply directly or indirectly an element of EnabBeh.

- **Enable list processing by scanning dependency relations.**

R4. The processing of Enable list is obtained, by scanning the Imply relation according to the dominate order, as follows:

1. For each element in the EnabAttr, not yet processed, find the Enabled Attributes and put them in EnabAttr.
2. For each element in the EnabAttr, not yet processed, find the Excluded Attributes and put them in DisabAttr.
3. For each element in the EnabDef, not yet processed, find the Enabled (1) definitions and put them in EnabDef, (2) functions and put them in EnabFun and (3) behaviors and put them in EnabBeh.
4. For each element in the EnabDef, not yet processed, find the Excluded definitions and put them in DisabDef.
5. For each element in the EnabFun, not yet processed, find the Enabled (1) definitions and put them in EnabDef, (2) functions and put them in EnabFun and (3) behaviors and put them in EnabBeh.
6. For each element in the EnabFun, not yet processed, find the Excluded functions and put them in DisabFun.
7. For each element in the EnabBeh, not yet processed, find the Enabled (1) definitions and put them in EnabDef, (2) functions and put them in EnabFun and (3) behaviors and put them in EnabBeh.
8. For each element in the EnabBeh, not yet processed, find the Excluded behaviors and put them in DisabBeh.

- **Disable List processing by scanning dependency relations.**

R5. The processing of Disable lists is obtained, by scanning the dependency relations according to the dominate order, as it follows:
1. For each element in the DisabAttr, not yet processed, find the Excluded organs and remove them from EnabAttr.
2. For each element in the DisabAttr, not yet processed, find the Disabled organs and put them in DisabAttr.
3. For each element in the DisabDef, not yet processed, find the Excluded definitions and remove them from EnabDef.
4. For each element in the DisabDef, not yet processed, find the Disabled (1) definitions and put them in DisabDef, (2) functions and put them in DisabFun and (3) behaviors and put them in DisabBeh.
5. For each element in the DisabFun, not yet processed, find the Excluded functions and remove them from EnabFun.
6. For each element in the DisabFun, not yet processed, find the Disabled (1) definitions and put them in DisabDef, (2) functions and put them in DisabFun and (3) behaviors and put them in DisabBeh.
7. For each element in the DisabBeh, not yet processed, find the Excluded behaviors and remove them from EnabBeh.
8. For each element in the DisabBeh, not yet processed, find the Disabled (1) definition sand put them in DisabDef, (2) functions and put them in DisabFun and (3) behaviors and put them in DisabBeh.

- **Loop on Enable list and Disable list processing.**

R6. Repeat R4 and R5 until all their elements are processed.

- **Final state**

R7. The result of this interpretation may be one of the following:
- A Failure, if coherence errors were detected.
- A View, if the interpretation successes. This View contains the Enabled Attributes, Attributes Definitions, Functions, and Behaviors.

4. **Evaluation**

We will evaluate our work by introducing the main contribution, and comparing it with related works, and we will present some of the areas where our work may be applicable.
4.1. Contribution

Inspired from genetics, our work aimed:
- Replace the “is-a” total inheritance hierarchy model (which is only an implementation model) used in current OOPL by the “composed by” selective inheritance model (which is a conceptual model) induced by the genetics; where there will only be one class “composed by other classes” that holds all properties associated to specific component instead of inheriting these properties from several classes and their ancestors. This has eliminated Complex Hierarchy by reducing the number of classes, where a genetic class is used.
- Formally and deeply model the selective inheritance based on the “composed by” class model.
- Use and develop the language extension proposed in [7] for the current OOPL so that we can implement our approach.

4.2. Related Works Comparison

Since there are several approaches for selective inheritance concept, each approach, as we mentioned, has been adopted from a different point of view, and also, each approach has its defects. In our approach we tried to overcome some of these defects. Table 1 shows a comparison -based on some significant criteria- between our proposed approach and the previously studied ones.

**Table 1. Comparison between our Approach and the Previous Approaches**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Criteria</th>
<th>Genetic class</th>
<th>Selection at class level</th>
<th>Properties classification</th>
<th>Conflict situation resolution</th>
<th>Selection at object level</th>
<th>Selection rules / control rules</th>
<th>Fully Bio-inspired</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.I at Class Level</td>
<td></td>
<td>×</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S.I at Object Level</td>
<td>Non-Genetic</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Genetic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our Approach</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3 Application Areas

Our concepts are closer to real life than the concepts adopted in current OOP, the number of classes is reduced, the variation in the same class is supported, and the inheritance is more powerful and practical than it is now in the Conventional OOP. This approach is needed in any application that significantly uses the inheritance, so, it can be applied to select the best object parameters in any object-oriented computer environment. As an example, in hardware field, (the micro-architecture is usually designed and tested with the aid of a software
simulator) where designing, testing, and producing a new computer processor is complex, the work [6] has proposed a specification of INTEL IA-32 using an architecture description language that selectively pre-determined the parent with the best fit to the object.

Also our approach may be applied in several works: Software Process Modeling, Software Reengineering, Software Reuse and relational databases. The results may be original and promising.

5. Conclusion

Through the study of the inheritance used in current OO, we found that it does not mimic the natural (real life) inheritance process as was claimed. In our approach, to eliminate all current OO inheritance’s problems, we used the Genetics concepts to model a selective inheritance that is closer to our natural life. In our model, an object can be created holding only desired and necessary properties and methods. This approach is young and new; it will be formalized and evaluated when it is widely accepted.

References

Authors

Darin Hamouda received her B.S. degree in Computer Science from Al-Zaytoonah University of Jordan and the M.S. degree in Computer Science from Philadelphia University, Jordan. She has been working as a Computer lab supervisor and an Assistant lecturer from 2005 to present. She can be reached at sad_sunset1@yahoo.com.

Said Ghoul has obtained his Master and Ph.D. Degrees, in Software Engineering, from University of Grenoble, France. His research interest includes Bio – inspired systems modeling and supporting concepts and methodologies. He is actually Full Professor at Philadelphia University - Faculty of Information Technology, Jordan. He can be reached at sghoul@philadelphia.edu.jo.

Hanan Hardan She is a Computer Science Lecturer at Philadelphia University. She received her MSc degree in Computer Science from the University of Jordan. She can be reached at hhardan@philadelphia.edu.jo.