International Journal of Engineering & Technology, 7 (2.20) (2018) 350-356

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Detecting Meta-Patterns from Frameworks Using Hybrid Genetic Algorithm

Tapan Kant[1]*, Manjari Gupta[2], Anil Kumar Tripathi[3], Meeta Prakash[4]

[1]*Department of Computer Science, Banaras Hindu University, Varanasi, India*
[2]*Department of Computer Science, Banaras Hindu University, Varanasi, India*
[3]*Department of Computer Science and Engineering, IIT BHU, Varanasi, India*
[4]*Infosys Ltd. India*
**Corresponding author E-mail:tapan.kant@gmail.com*

Abstract

Meta-patterns are a sort of basic object-oriented constructs that have been used to design an object-oriented framework. It has been used to precisely describe possible design pattern of a framework at meta-level to manifest framework hot-spots and its corresponding adaptability. The present study is an attempt to develop a genetic algorithm approach for detecting the types and numbers of meta-patterns. For this purpose we have converted the UML class diagram of object-oriented framework and meta-patterns into directed graph and applied hybrid genetic algorithm. The obtained results from the proposed algorithm are further validated manually with the recall and precision percentage of 86.20 and 80.64 respectively. Overall the study demonstrates the utility of the uniquely proposed algorithm for the near accurate identification of meta-patterns for high reusability. This can be applied on frameworks for assessing the evolution process, documentation of hot-spots and reducing the customization effort.

Keywords: *Genetic Algorithm; meta-patterns; Object-oriented framework; Sub-graph.*

1. Introduction

The main aim of object-oriented software engineering is to improve the quality of the software through reuse. Among the reuse techniques, the object-oriented framework provides flexible architecture and reusable design for specific domain and thus is widely accepted as a dominant technique in software development. It offers predefined adaptation point that is known as hot-spots [1]. Hot-spots consist of small structures to acquire object-oriented reusable design. These small reusable structures are often termed as meta-patterns [2]. Template and hook methods can be grouped into a single class or group of classes together with their interaction often termed as meta-patterns.

Meta-pattern [2] is as an approach to develop framework independent to specific domain and can also be used to precisely describe possible design pattern of a framework at meta-level, to manifest framework hot-spots and its corresponding adaptability. Schauer et. al. [3] detected the meta-patterns and described that a set of hook methods together with their associated template methods, in which all the hook methods that are called by same set of template methods are captured into one single hot-spot. Further, Thummalapenta and Xie [4] also detected and utilized the meta-patterns for the hot-spots detection. These reports suggested that the meta-patterns can be utilized for the hot-spots documentation, assessing the evolution of the framework, and framework based application development. However, these techniques are based on source code and limited to particular programming language. Therefore, there is requirement to develop a technique which is more direct and language independent for enhanced reusability Graphs are increasingly being used as dominant technique to model data representation such as chemical structures, computer networks, digital circuits, etc. The work in [5][6][7][8] uses graphs to represent UML class and object diagrams. Further, there are several reports which usage sub-graphs isomorphism to detect design patterns from object-oriented software. Sub-graph isomorphism may be formulated as a decision problem where input is a pair of graphs G and G1, and the output is yes if G1 is isomorphic to a sub-graph of G and no otherwise. Finding sub-graph of G that is isomorphic to G1 is corresponding optimization problem and NP-Complete. Ullmann [9] proposed backtracking procedure to solve sub-graph isomorphism problem in exponential time complexity. Further, two algorithms, the Durand-Pasari algorithm [10] and McGregor algorithm [11] are proposed to solve maximum common sub-graph [12][13]. A Hybrid genetic algorithm was proposed to solve the sub-graph isomorphism problem [14]. Genetic Algorithms (GA) are inspired from the processes of natural evolution. GA is an adaptive heuristic search algorithm which uses evolutionary facts of natural selection and genetics. GA makes use of historical information to perform the search into a region of better performance within the search space. To solve a problem, GA simulates the survival of the fittest among individuals over several generations. The chromosomes of each generation comprises of a population of character strings. Each individual correspond to a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution.

Therefore in the present study, we propose a unique hybrid genetic algorithm for detection of meta-patterns. The problem of meta-pattern detection is mapped into the problem of sub-graph isomorphism. The process of obtaining meta-patterns of a framework through sub-graph isomorphism has been shown in this work by : i) converting a system design UML diagram into a graph and ii) applying hybrid genetic algorithm to solve the problem of sub-graph isomorphism

In the process of application of hybrid genetic algorithm, we have modified the fitness function so that it can be applied for meta-pattern detection. This is being demonstrated through Java implementation of the proposed hybrid genetic algorithm for detecting meta-patterns for four widely used open source frameworks. We have manually validated our result to measure the performance of our approach.

The rest of the paper is organized as follows. Section II establishes the background of our work. We explain the graph representation of UML diagrams of meta-patterns as well as of framework design in section III. In section IV, we explained how to apply the hybrid genetic algorithm to sub-graph isomorphism problem. In the next section V we discuss the results and scope of future works. Related work in this area is described in detail in section VI. The last section VII covers the conclusion.

2.Background

This section describes meta-patterns and their corresponding graphs to establish the background of our work.

2.1 Meta-Pattern an Overview

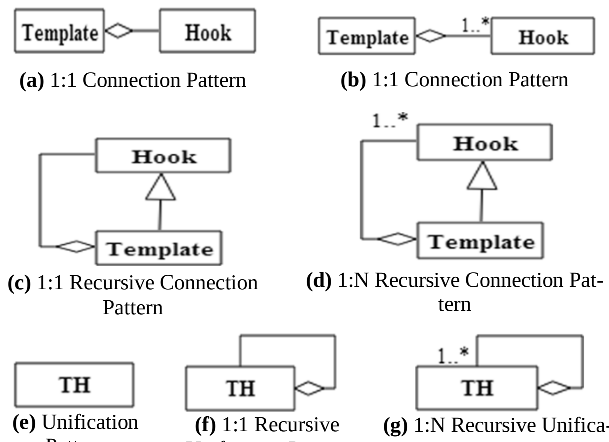
The core concept of meta-patterns manifest framework design and adaptability. The fine grain structures of meta-patterns are template and hook methods. The classes which contain template methods are known as template class whereas the classes which contain hook methods are called hook class. According to Pree [2] patterns are elegant and powerful approach to describe framework centred design patterns at meta-level. Meta-patterns can be used as an approach to develop frameworks independent of specific domain. The fundamental of meta-pattern is based on Open-closed principle. The Open-closed principle states that a software module should be opened for extension and closed for modification. The object-oriented frameworks follow the Open-closed principle and provides flexible and reusable design. The object-oriented frameworks provide predefined extension point known as hot-spots. Hot-spots consist of small structures to acquire object-oriented reusable design. These small reusable structures are often termed as meta-patterns. Template and hook methods can be grouped into either in a single class or group of classes together with their interaction often termed as meta-patterns. Stating this more objectively, we can say that:

(i). Template methods are concrete, reckoned as complex and determine generic behaviour between objects by calling hook methods. Hook methods act as proxy, usually considered as mutable part (in framework); it could be concrete or abstract invoked by a single template method or group of template methods.

(ii). Template methods call hook methods, template and hook classes are combined in particular fashion to form meta-patterns either through association relationship or through inheritance relationship.

(ii). In association relationship between template and hook classes, a single object of template class could be associated with a single object or more than one objects of the hook class. In inheritance relationship, template class and hook class have parent and child relationship where hook class is parent and child is template class.

Moreover, there are seven different types of meta-patterns as defined by Pree in [2]. The interaction among template & hook methods either within a class or group of classes is shown in Figure 1. The basis of meta-pattern interaction can be either through unification, abstract coupling or recursive structures [2]. The examples of meta-patterns having template and hook methods in a same class is known as Unification meta-patterns shown in figure 1e, 1f & 1g. The meta-patterns which rely on abstract coupling is known as connection patterns in which an object of template class refers either exactly one object or any number of objects, of hook class and represented as 1:1 Connection Pattern (Fig. 1a) a 1:N Connection Pattern (Fig. 1b), respectively. The example of meta-patterns which are based on recursive structures is known as recursive connection pattern in which template class is a descendant of hook class also maintains one or any number of reference(s) to its ancestor class and represented as 1:1 Recursive Connection Pattern (Fig. 1c) and 1:N Recursive Connection Pattern (Fig. 1d), respectively. In Recursive Connection Pattern, the template and hook methods have usually the same name. The following example (Fig. 2) is helpful to better understand meta-patterns.



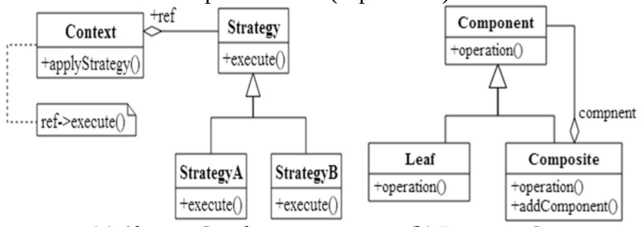
(a) 1:1 Connection Pattern **(b)** 1:1 Connection Pattern

(c) 1:1 Recursive Connection Pattern **(d)** 1:N Recursive Connection Pattern

(e) Unification Pattern **(f)** 1:1 Recursive Unification Pattern **(g)** 1:N Recursive Unification Pattern

Fig. 1: UML diagram of meta-pattern as in [2]

In figure 2, we describe strategy design pattern and composite design pattern which is concrete example of abstract coupling and recursive structure, respectively. In the given example of strategy design pattern (Fig. 2a); the Context class contains a reference to an abstract class (Strategy). Here, the *applyStrategy()* method of Context class is template methods and the *execute()* method of Strategy class is the hook method. Context class is abstractly coupled with Strategy class that allows hard-wiring between these classes. The example of composite pattern is given in figure 2b, the *operation()* method of Composite class is template and the *operation()* method of Component class is hook. The recursive structure enables Composite class (the descendant class) to contain the reference of Component class (super class).



(a) Abstract Coupling **(b)** Recursive Structure

Fig 2: Example of meta-patterns based on abstract coupling & recursive structure

In case of unification meta-pattern adaptation is done at compile time since a subclass is required to be created and hook method is overridden there. While in case of connection meta-patterns adaptation is done at run-time since subclasses of hook class are need to be defined, instantiated and plugged into template objects at run time.

2.2. Class Relationship Representation through Graph

The class diagram of an object-oriented design under study can be represented through graph. Pande et. al. [15] shows that how class diagram can be represented through digraph. Classes are represented by nodes and relationships (i.e. generalization, aggregation or association) are represented by labelled directed edges.

2.3. Graph Representation of System Design and Meta-Patterns

In this approach, the UML class diagrams of system design and meta-patterns are converted into digraphs. We use the term target graph for digraph of framework design and pattern graph for digraph of a meta-pattern. A node of the graph represents a class in the UML class diagram and relationships among classes are represented by edges. Each node and edge is labelled. The label of each node is abstract/concrete. More labels of classes can be taken to include more other attributes of a class that will improve the method of meta-pattern detection. In this preliminary effort, the attributes (abstract/concrete) are only considered for a class. The edges which represent the relationships generalization, aggregation, & one-to-many association, among classes are labelled 1, 2, or 3 respectively. In additional to this, important information concerned to meta-patterns is classes having template and hook methods. We are incorporating this information in the graph representation by adding one more edge from a class having temple method to a class having hook method. The label of this edge is 4. In figure 3, the design of JUnit framework (a de facto standard unit testing framework for Java) is represented by UML class diagram, is shown. The graph corresponding to the framework is shown in figure 4. Similarly, the graph corresponding to the meta-pattern (Figure 1) is shown in figure 5.

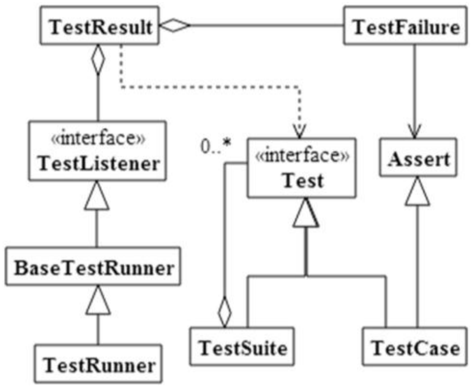


Fig. 3: UML Diagram - JUnit Framework

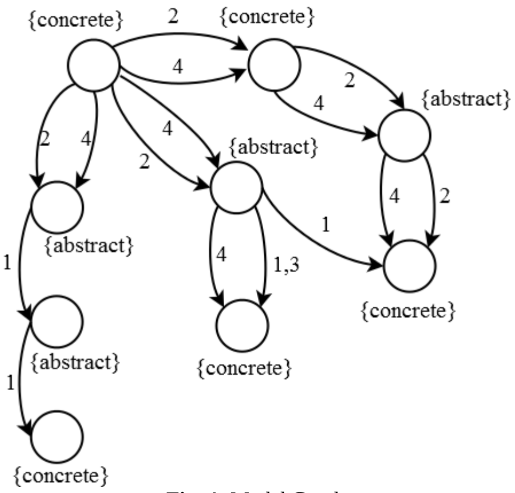


Fig. 4: Model Graph

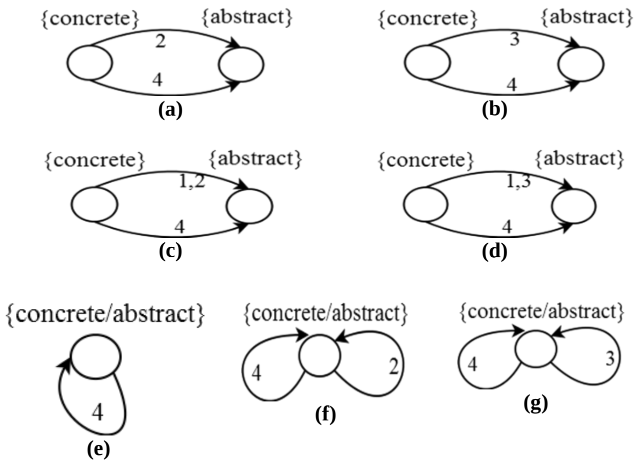


Fig. 5: Model Graph of Meta-pattern as Shown in Figure 2

3. Hybrid Genetic Algorithm Based Solution of Sub-Graph Isomorphism Problem

The proposed Hybrid genetic algorithm (HGA) has been used for the problem of sub-graph isomorphism is described here. HGA generates a set of initial solutions that evolve over generations and terminates on stopping condition. This genetic algorithm outputs the set of possible meta-patterns in a framework. These identified meta-patterns are verified using the following rules [16].

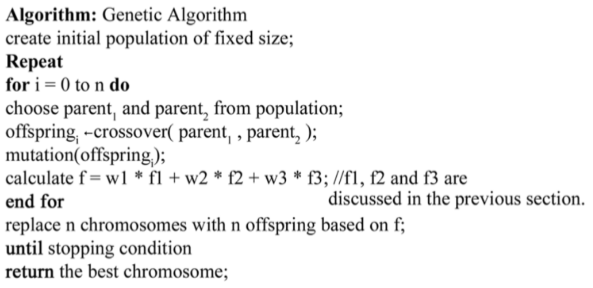
1. The identified unification (and recursive unification) meta-patterns in which hook method are overridden in subclasses is true unification meta-patterns and the remaining meta-patterns not satisfying this condition are discarded. This implies that in case of true unification meta-patterns hook methods are pure abstract methods.

2. The identified connection (connection and recursive-connection) meta-patterns in which subclasses of hook class are defined, instantiated and plugged into template objects at run time are true connection meta-patterns and the remaining not satisfying this condition are discarded.

The tool we developed based on this proposed approach would correctly identify unification meta-patterns but the identified connection meta-patterns need to be further cross-checked manually since few of the identified connection meta-patterns may not be true meta-patterns.

3.1. Hybrid Genetic Algorithm

The HGA approach that we have applied in our work is represented below. As explained above, two graphs are taken such that one corresponds to the framework and other corresponds to the meta-pattern. A large number of random populations are generated and mapping between these two graphs are performed. Different mappings are corresponding to different possible solutions. In order to obtain best mapping on the basis of their fitness function genetic algorithm is applied over these random mapping. The termination condition of our hybrid genetic algorithm is number of generation. The results in this paper consider 500 generations. A population of size 1000 is considered.



```
Algorithm: Genetic Algorithm
create initial population of fixed size;
Repeat
for i = 0 to n do
    choose parent_1 and parent_2 from population;
    offspring_i ←crossover( parent_1 , parent_2 );
    mutation(offspring_i);
    calculate f = w1 * f1 + w2 * f2 + w3 * f3; //f1, f2 and f3 are
end for                                          discussed in the previous section.
replace n chromosomes with n offspring based on f;
until stopping condition
return the best chromosome;
```

Fig. 6: Hybrid Genetic Algorithm

3.2. Chromosome Representation

A chromosome represents node to node mappings of pattern graph to target graph. Nodes of both graphs are numbered. The proposed chromosome is a one dimensional array of length two: C(2). Length of this array is corresponding to number of nodes in the pattern graph. As described in section 2, the graph corresponding to any meta-pattern has only two nodes hence the size of a chromosome is two. The value at the first index of chromosome is the node of the target graph mapped to the first node of the pattern graph. Similarly the value at the second index of chromosome is a node in the target graph mapped to second node of the pattern graph.

3.3. Fitness Function

The state-of-the-art of evolutionary or heuristic algorithms to solve a problem is optimal. In order to find an effective search path we have chosen appropriate fitness function. In this algorithm, fitness function has been used to map the number of different edges between two graphs to solve the sub-graph isomorphism problem [17]. Let $\boldsymbol{G_1} = (\boldsymbol{V_1}, \boldsymbol{E_1}), \boldsymbol{G_2} = (\boldsymbol{V_2}, \boldsymbol{E_2})$, where $|\boldsymbol{V1}| \le |\boldsymbol{V2}|$, and $\boldsymbol{G_s} = (\boldsymbol{V_s}, \boldsymbol{E_s})$ be a sub-graph of $\boldsymbol{G_2}$ mapped to $\boldsymbol{G_1}$. Then

$$\boldsymbol{f} = \sum_{e \in E_1} \boldsymbol{I}(\boldsymbol{e}, \boldsymbol{E_s}) + \sum_{e \in E_s} \boldsymbol{I}(\boldsymbol{e}, \boldsymbol{E_1}) \quad (1)$$

where,

$$\boldsymbol{I}(\boldsymbol{e}, \boldsymbol{E}) = \begin{cases} \boldsymbol{0}, \boldsymbol{if}\ \boldsymbol{e} \in \boldsymbol{E} \\ \boldsymbol{1}, \boldsymbol{ohterwise} \end{cases}$$

If the value of $\boldsymbol{f}$ is $\boldsymbol{0}$, it means that $\boldsymbol{G_1}$ is a complete sub-graph of $\boldsymbol{G_2}$. For meta-pattern detection only the $\boldsymbol{1^{st}}$ part of this fitness function $\boldsymbol{f}$ is useful. The second component we cannot include since there may be some edges in the sub-graph of model graph but may not be there in the pattern graph and even then this pattern graph may exists in the target graph. Thus the first component of our fitness function that we use in this work is

$$\boldsymbol{f1} = \sum_{e \in E_1} \boldsymbol{I}(\boldsymbol{e}, \boldsymbol{E_s}) \quad (2)$$

The number of different edges as a whole is computed by the first component of the fitness function. One can exclude the unattractive candidate permutations in a heuristic by taking advantage of the degrees of nodes. For example, if the degrees of the two nodes are unequal then the node of one graph cannot be mapped onto the node of the other graph. This property has been extended in [17]. A pair of nodes, each from one of the two graphs is improbable to match if the node from the larger graph has a lower degree than the one from the smaller graph. Therefore, the comparison of degrees of any two nodes which are mapped to each other can be used differently to measure the fitness of a solution. Thus for $\boldsymbol{g}: \boldsymbol{V}(\boldsymbol{G_1}) \Rightarrow \boldsymbol{V}(\boldsymbol{G_2})$ we include function described below as a second component of our fitness function.

$$\boldsymbol{f2} = \sum_{v \in V(G_1)} \boldsymbol{J}(\boldsymbol{v}) \quad (3)$$

where,

$$\boldsymbol{J}(\boldsymbol{v}) = \begin{cases} \boldsymbol{0}, \boldsymbol{if}\ \boldsymbol{d_{out}}(\boldsymbol{g}(\boldsymbol{v})) \ge \boldsymbol{d_{out}}(\boldsymbol{v})\ \boldsymbol{and}\ \boldsymbol{d_{in}}(\boldsymbol{g}(\boldsymbol{v})) \ge \boldsymbol{d_{in}}(\boldsymbol{v}) \\ \boldsymbol{1}, \boldsymbol{otherwise} \end{cases}$$

The graph considered in the both of these papers [14][17] are not labelled graph. They give fitness function for simple directed graph. We have represented framework design and meta-patterns by labelled graphs. Thus for our problem only the above defined two components of fitness functions are not sufficient. Fitness function must include labels of vertices as well as labels of edges. Edge labels can be matched with fitness function $\boldsymbol{f1}$ itself. The definition of $\boldsymbol{f1}$ is little bit changed from the definition of first part of $\boldsymbol{f}$. Here, to match edge we will consider label of that edge. That is $\boldsymbol{I}(\boldsymbol{e}, \boldsymbol{E_1})$ for all $\boldsymbol{e} \in \boldsymbol{E_s}$ will give the value $\boldsymbol{0}\ \boldsymbol{if}\ \boldsymbol{e} \in \boldsymbol{E_1}$ and labels of these two edges are also same. Vertex labelling is another important information that must also be compared to match the mapped nodes. To incorporate labels of vertices we propose another component of fitness function: The function $\boldsymbol{f3}$ measured differently by comparing the labels of any two nodes that are mapped to each other.

$$\boldsymbol{f3} = \sum_{v \in V(G_1)} \boldsymbol{K}(\boldsymbol{v}) \quad (4)$$

where,

$$\boldsymbol{K}(\boldsymbol{v}) = \begin{cases} \boldsymbol{0}, \boldsymbol{if}\ \boldsymbol{Label}(\boldsymbol{g}(\boldsymbol{v})) \subseteq \boldsymbol{Label}(\boldsymbol{v}) \\ \boldsymbol{1}, \boldsymbol{otherwise} \end{cases}$$

Thus our fitness function is $\boldsymbol{f1} + \boldsymbol{f2} + \boldsymbol{f3}$.

4. Results and Discussion

This section discusses the analysis of the results obtained by our approach and also describes the impact of our approach on object-oriented frameworks. We have not found any earlier reported work in detecting meta-patterns. Therefore, we have developed a tool for detection of meta-patterns.

4.1. Analysis of Results

Table 1 shows four widely utilized open source object-oriented frameworks which have been used as experimental subjects in our experiment. The columns of the table are subject, version, the total numbers of classes, the total number of methods and the URL of the frameworks.

Table 1: Subject used in experiment

Subject	Version	Total Number of Classes	Total Number of Methods	URL
JUnit	3.4	31	319	www.junit.org
JHotDraw	7.0.6	196	2386	www.http://jhotdraw.org
BCEL	5.2	335	2935	jakarta.apache.org/bcel
Open-JGraph	0.9.0	181	1084	open-jgraph.sourceforge.net

The Table 2 shows our results on the above described object-oriented frameworks. The columns of table 2 are subject, technique used, types of meta-patterns and total number of meta-patterns. There are three rows per subject. The first row shows meta-patterns identified by genetic algorithm. The second row shows meta-patterns identified after applying rule 1 & 2 (as discussed in section 3) which are used to refine the output of genetic algorithm. The third row shows manually identified meta-patterns to validate the results obtained by our approach (shown in row 2). We have performed manual validation for Junit framework only.

The numbers of classes in other three frameworks are very high thus we could not perform manual validation of these frameworks. The symbol "-" in manually row indicate that manual validation is not performed.

However, we have also identified 6 false positive and 4 false negative meta-patterns. The false positive and negative meta-patterns fall in the template and hook class are shown in table 3. These results can be purposefully utilized in hot-spot detection and documentation of frameworks as well as in design pattern detection. The documentation of hot-spots with different types of meta-patterns would be helpful for application developers in understanding the design and structure of frameworks. This will reduce the customization effort made by application developers. Further, these results can also be applied in assessing the evolution of a framework. As the number of connection and recursive-connection meta-patterns increase clearly reflect that the framework is growing towards maturity [2].

The reason behind false positive identification of meta-pattern is indirect hierarchical relationship among classes. Further due to the implementation and language limitation in Junit 3.4, the tool has detected false negative. The Junit version 3.4 was developed using Java programming language of version 1.3. This version of Java does not support generic programming. Therefore in order to check the accuracy of the tool we have also applied this tool on Junit version 4.10 on which our approach does not found any false negative. The Junit version 4.10 was developed using Java programming language of version 1.4 that supports generic programming therefore the tool does not detect false negative. The recall percentage of our algorithm is 86.20 and the precision percentage is 80.64. The obtained results clearly indicate that the proposed tool can be applied for correctly identification of meta-patterns. The impact of our approach on object-oriented application frameworks has been discussed in following subsections.

4.2. Impact of Our Approach on Object-Oriented Framework

Meta-patterns are elegant and powerful approach which helps in developing object-oriented frameworks independent to a specific domain. Meta-patterns play important role in the construction of frameworks hot-spots and also in frameworks adaptation. We have obtained different types of meta-patterns, i.e. 22, 2, 1, connection, unification and recursive connection meta-patterns from Junit-3.4 framework. Moreover, the results obtained by our approach can be utilized for assessing the evolution of the framework, hot-spots documentation and reducing the customization effort made by application developers, which have been discussed below.

4.2.1. Assessing Evolution of Framework

A framework is a collection of abstract classes and their collaborators. Frameworks are larger in size therefore development cost is high. The management has to keep tracking to assess, the evolution of framework from historical information. Nowadays, historical information is being used to assess the evolution-proneness of application frameworks [18]. According to Mattsson and Bosch [18], historical information gathered from the framework versions can be used to assess the framework's evolution in terms of size, growth and change rate. Application framework tends to evolve from white-box to black-box [19]. Frameworks are instantiated either through inheritance or by composition mechanism. The framework that uses inheritance mechanism for instantiation is white-box framework and composition mechanism is used in black-box [2]. The author also described that the unification meta-patterns are involved in white box frameworks and connection and recursive connection meta-patterns are involved in black box framework.

The detection of type of meta-pattern is necessary to evaluate the maturity level of the framework. In this regard we have developed a tool to identify the types of meta-patterns and detected 25, 5, 1 connection, unification and recursive connection meta-patterns. This depicts that our approach can be used to assess the evolution of framework.

4.2.2. Design Pattern Identification

The design patterns describe the general solution to a commonly occurring problem [20]. The meta-patterns detection process can be useful in identification of design pattern. According to Pree the design patterns Abstract Factory, Command, Interpreter, Observer, Prototype, Builder, State and Strategy are based on connection meta-patterns. Moreover, the design patterns that are based on recursive connection meta-patterns are Composite, Decorator and Chain-of-Responsibility. Thus, identification of connection and recursive connection meta-patterns using proposed approach will lead to the identification of design patterns used in the frameworks.

4.2.3. Framework documentation

Frameworks are designed to solve large-grain problem therefore it is larger in size and highly complex. The learning curve of the framework is very high. A good documentation is needed for framework to easily understand and use it. The framework documentation should describe the purpose or domain, for which it has been developed; a description – how to use the framework; and structural design and its behaviour of the framework. These descriptions are broadly categorised as Purpose, Usages and Design [21]. The architectural design of the frameworks were the primary focus of many works [20][22][23][24]. Only some report [25][26] which deals in the intended use and the purpose of the frameworks. Another similar approach has been proposed in [27] - a method to document the design of the frameworks by means of “Hooks”. It provides an alternative view of the design which shows how and where changes can be made to adapt a framework. The report in [28] detected hot-spots through UsageMetrics and shown total of 14 classes (8 template classes and 6 hook classes) involved in hot-spots hierarchy. The proposed algorithm also identifies those classes as template and hook. This indicates that our tool can be used to documentation of the framework and can also be used in detecting hot-spots.

4.2.4 Customization effort

Framework instantiation is a process of creating application through framework. In this process, the flexible parts (called hot-spots) of the framework are adapted according to specific need. In order to instantiate a framework, an application developer has to learn the steep instantiation process [18], usually it takes 6 to 12 months to become productive [29]. The flexible architecture of the framework makes hard to comprehend and the lack of high-level documentation to developers make difficult to instantiate. In order to instantiate a framework easily, the design hints of flexible parts should be available to the developers. An approach presented by Froehlich et. al. [27 to instantiate a framework through “hooks”. Each hook is an explicit solution described in different sections of structured and textual language. By combining several hooks on can achieve the solution for a large problem. Our proposed algorithm can be used to describe hooks. The hybrid genetic algorithm have been proven to be useful for the detection of maximum common sub-graph or chemical structure matching [30][31]. We have implemented our algorithm in java programming language and applied on four different open source frameworks shown in table 1.

Table 2: Types of meta-patterns identified in different types of frameworks

		Type of Meta-patterns							
Subject	**Technique Used**	**Number of 1:1 Connection**	**Number of 1:N Connection**	**Number of 1:1 Recursive Connection**	**Number of 1:N Recursive Connection**	**Number Unification**	**Number 1:1 Unification**	**Number of 1:N Unification**	**Total**
Junit	GA	27	1	1	1	0	1	0	31
Rule 1 & 2	27	1	1	1	0	1	0	31	
Manually	22	0	1	1	0	1	0	25	
JHotDraw	GA	199	12	8	6	9	3	3	240
Rule1 & 2	192	12	3	6	8	3	2	226	
Manually	-	-	-	-	-	-	-	-	
BCEL	GA	115	26	22	5	1	3	1	173
Rule1 & 2	112	26	18	4	1	2	1	164	
Manually	-	-	-	-	-	-	-	-	
OpenJGraph	GA	74	8	6	2	0	4	2	96
Rule1 & 2	74	8	4	1	0	4	2	93	
Manually	-	-	-	-	-	-	-	-	

Table 3: Meta-patterns identified as false positive and false negative

	Classes Involved		Reason
	Template Class	Hook Class	
False Positive	junit.awtui.TestRunner	junit.extensions.TestDecorator	Indirect Hierarchical Relationship
	junit.extensions.TestDecorator	junit.framework.Test	
	junit.framework.TestFailure	junit.framework.TestSuite	
	junit.swingui.TestBrowser	junit.extensions.TestDecorator	
		junit.extensions.TestDecorator	
	junit.framework.TestResult	junit.swingui.TestTreeModel	
		junit.framework.TestFailure	
False Negative	junit.awtui.TestRunner	junit.framework.Test	Language Limitation and Implementation in code
	junit.framework.TestSuite	junit.framework.Test	
		junit.framework.TestListener	

5. Related work

Many researchers are working in the field of design pattern mining. Pande et. al. [15] used graph theoretical approach to model design patterns into graph and applied decomposition algorithm to detect design patterns. Bernardi et al. [32] proposed an approach by tracing systems' source code components with the roles they play in the patterns in which design patterns are modelled based on their high-level structural properties. Their approach has limitation in detecting complex designs and architectural patterns. On the basis of similarity scoring between graph vertices a design pattern detection methodology was proposed in [33]. The design patterns that are modified from their standard representation are also recognized by this approach. The attributed relational graph is used to describe design patterns and system design and sub-graph isomorphism approach is applied for the detection process of design patterns in [34]. Machine learning techniques are also being used for mining design pattern. Ferenc et. al. [35] used machine learning to enhance pattern mining by filtering out as many false hits as possible, by providing true and false pattern instances of a learning database. Zanoni et. al. [36] also used machine learning methodology for design pattern detection. Design patterns and meta-patterns both are essential parts of a framework. We could not find any work to detect meta-patterns from framework or in general from an object oriented software.

There are few works on hot spots detection from frameworks. A reverse engineering technique to model framework reuse interfaces from implementation is introduced in [37]. Further, an Eclipse plug-in (called FrUiT) [38] is developed to extract reusable patterns from existing framework instantiation. In this approach the data mining techniques has been used. This report also presents a first assessment by mining parts of the Eclipse framework. A tool to recover hot-spots automatically from the source code of the frameworks have been proposed by Schauer et al. in [3] and described hot-spots as composition hot-spot \& inheritance hot-spot. Moreover, A tool called SpotWeb has been proposed in [28] to detect hot-spots and cold-spots in a given framework. This tool usages code-search-engine-based approach to detect hot-spots and cold-spots by mining code examples from open source repositories available on the web. The problems in instantiating frameworks through explicit contract by means of meta-patterns which require development environment and vocabulary of code of meta-patterns has been addressed in [39]. Likewise, the problems concerned with framework augmentation and techniques to explicit framework reuse has been presented in [40][41][42][43].

6. Conclusion

This paper proposes a unique meta-pattern detection mechanism based on a hybrid genetic algorithm for sub-graph isomorphism problem by specifying a new fitness function. The observed results with the recall and precision percentage of 86.20 and 80.64 respectively, clearly reflect that the proposed algorithm can be utilized for the detection of meta-patterns. The approach we used for meta-pattern detection can be extended for framework based application development where application developers can easily identify the possible structure of classes and glue codes specific to their domain. This approach can be used in the object-oriented software engineering to minimize the framework customization effort and improve the quality of documentation. The further work and future expansion of this topic will involve more extensive experiments on the various types and size of frameworks.

References

[1] W. Pree, "Hot-spot-driven framework development," in Summer School on Reusable Architectures in Object-Oriented software Development, pp. 123-127, ACM, 1995.

[2] W. Pree, "Meta patterns - a means for capturing the essentials of reusable object-oriented design," in *Object-Oriented Programming* (M. Tokoro and R. Pareschi, eds.), vol. 821 of *Lecture Notes in Computer Science*, pp. 150-162, Springer Berlin Heidelberg, 1994.

[3] R. Schauer, S. Robitaille, F. Martel, and R. Keller, "Hot spot recovery in object-oriented software with inheritance and composition template methods," in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pp. 220-229, 1999.

[4] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 327{336, Sept 2008.

[5] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, "Attributed graph transformation with node type inheritance," *Theoretical Computer Science*, vol. 376, no. 3, pp. 139 -163, 2007. Fundamental Aspects of Software Engineering.

[6] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer, "Integrating metamodelling aspects with graph transformation for efficient visual language definition and model manipulation," in *Fundamental Approaches to Software Engineering* (M. Wermelinger and T. Margaria-Steffen, eds.), vol. 2984 of *Lecture Notes in Computer Science*, pp. 214{228, Springer Berlin Heidelberg, 2004.

[7] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski, "An integrated semantics for uml class, object and state diagrams based on graph transformation," in *Integrated Formal Methods* (M. Butler, L. Petre, and K. Sere, eds.), vol. 2335 of *Lecture Notes in Computer Science*, pp. 11-28, Springer Berlin Heidelberg, 2002.

[8] A. Maraee and M. Balaban, "Efficient reasoning about finite satisfiability of uml class diagrams with constrained generalization sets," in *Model Driven Architecture- Foundations and Applications* (D. Akehurst, R. Vogel, and R. Paige, eds.), vol. 4530 of *Lecture Notes in Computer Science*, pp. 17-31, Springer Berlin Heidelberg, 2007.

[9] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, pp. 31-42, Jan. 1976.

[10] P. J. Durand, R. Pasari, J. W. Baker, and C.-c. Tsai, "An efficient algorithm for similarity analysis of molecules," *Internet Journal of Chemistry*, vol. 2, no. 17, pp. 1-16, 1999.

[11] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23-34, 1982.

[12] F. P. Conte Donatello and V. Mario, "Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs.," *Journal of Graph Algorithms and Applications*, vol. 11, no. 1, pp. 99-143, 2007.

[13] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento, "A comparison of algorithms for maximum common subgraph on randomly connected graphs," in *Structural, Syntactic, and Statistical Pattern Recognition* (T. Caelli, A. Amin, R. Duin, D. de Ridder, and M. Kamel, eds.), vol. 2396 of *Lecture Notes in Computer Science*, pp. 123-132, Springer Berlin Heidelberg, 2002.

[14] K. Kim and B.-R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, CO '10, (New York, NY, USA), pp. 1211{1218, ACM, 2010.

[15] A. Pande, M. Gupta, and A. Tripathi, "A new approach for detecting design patterns by graph decomposition and graph isomorphism," in *Contemporary Computing* (S. Ranka, A. Banerjee, K. Biswas, S. Dua, P. Mishra, R. Moona, S.-H. Poon, and C.-L. Wang, eds.), vol. 95 of *Communications in Computer and Information Science*, pp. 108-119, Springer Berlin Heidelberg, 2010.

[16] W. Pree, "Hot-spot-driven framework development," in *Summer School on Reusable Architectures in Object-Oriented software Development*, pp. 123-127, ACM, 1995.

[17] J. Choi, Y. Yoon, and B.-R. Moon, "An efficient genetic algorithm for subgraph isomorphism," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, (New York, NY, USA), pp. 361-368, ACM, 2012.

[18] M. Mattsson and J. Bosch, "Observations on the evolution of an industrial oo framework," in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pp. 139-145, 1999.

[19] M. Mattsson and J. Bosch, "Characterizing stability in evolving frameworks," in *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*, pp. 118-130, Jul 1999.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "*Design patterns: elements of reusable object-oriented software"*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] R. E. Johnson, "Documenting frameworks using patterns," in *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, (New York, NY, USA), pp. 63-76, ACM, 1992.

[22] R. Campbell and N. Islam, "A technique for documenting the framework of an object-oriented system," in *Object Orientation in Operating Systems, 1992., Proceedings of the Second International Workshop on*, pp. 288-300, Sep 1992.

[23] K. Beck and R. E. Johnson, "Patterns generate architectures," in *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP '94, (London, UK, UK), pp. 139-149, Springer-Verlag, 1994.

[24] D. B. Lange and Y. Nakamura, "Interactive visualization of design patterns can help in framework understanding," in *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, (New York, NY, USA), pp. 342-357, ACM, 1995.

[25] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *J. Object Oriented Program.*, vol. 1, pp. 26-49, Aug. 1988.

[26] I. ParcPlace-Digitalk, *VisualWorks: Cookbook*. ParcPlace-Digitalk, Incorporated, 1995.

[27] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson, "Hooking into object-oriented application frameworks," in *ICSE '97: Proceedings of the 19th international conference on Software engineering*, (New York, NY, USA), pp. 491-501, ACM, 1997.

[28] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, (New York, NY, USA), pp. 204-213, ACM, 2007.

[29] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, "*Implementing application frameworks: objectoriented frameworks at work*." New York, NY, USA: John Wiley & Sons, Inc., 1999.

[30] R. D. Brown, G. Jones, P. Willett, and R. C. Glen, "Matching two-dimensional chemical graphs using genetic algorithms," *Journal of Chemical Information and Computer Sciences*, vol. 34, no. 1, pp. 63-70, 1994.

[31] M. Wagener and J. Gasteiger, "The determination of maximum common substructures by a genetic algorithm: Application in synthesis design and for the structural analysis of biological activity," *Angewandte Chemie International Edition in English*, vol. 33, no. 11, pp. 1189-1192, 1994.

[32] M. Bernardi, M. Cimitile, and G. Di Lucca, "A model-driven graph-matching approach for design pattern detection," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 172-181, Oct 2013.

[33] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 896-909, Nov 2006.

[34] L. Qing-hua, Z. Zhi-xiang, and B. Ke-rong, "Design pattern mining using graph matching," *Wuhan University Journal of Natural Sciences*, vol. 9, no. 4, pp. 444-448, 2004.

[35] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 295-304, Sept 2005.

[36] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102 -117, 2015.

[37] J. Viljamaa, "Reverse engineering framework reuse interfaces," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, (New York, NY, USA), pp. 217-226, ACM, 2003.

[38] M. Bruch, T. Sch¨afer, and M. Mezini, "Fruit: Ide support for framework understanding," in *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, (New York, NY, USA), pp. 55-59, ACM, 2006.

[39] T. Tourwe and T. Mens, "Automated support for framework-based software," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 148-157, Sept 2003.

[40] G. Kiczales and J. Lamping, "Issues in the design and specification of class libraries," in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '92, (New York, NY, USA), pp. 435-451, ACM, 1992.

[41] J. Lamping, "Typing the specialization interface," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, (New York, NY, USA), pp. 201-214, ACM, 1993.

[42] M. Mezini, "Maintaining the consistency of class libraries during their evolution," *SIGPLAN Not.*, vol. 32, pp. 1-21, Oct. 1997.

[43] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse contracts: Managing the evolution of reusable assets," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, (New York, NY, USA), pp. 268-285, ACM, 1996.