Sub-graph Mining: Identifying Micro-architectures in Evolving Object-oriented Software

Ahmed Belderrar, Segla Kpodjedo, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Philippe Galinier
SOCCER Lab. & Ptidej Team
DGIGL, École Polytechnique de Montréal,
Montréal, Canada
{*ahmed.belderrar,segla.kpodjedo,yann-gael.gueheneuc,giuliano.antoniol,philippe.galinier*}*@polymtl.ca*

***Abstract*—Developers introduce novel and undocumented micro-architectures when performing evolution tasks on object-oriented applications. We are interested in understanding whether those organizations of classes and relations can bear, much like cataloged design and anti-patterns, potential harm or benefit to an object-oriented application. We present SGFinder, a sub-graph mining approach and tool based on an efficient enumeration technique to identify recurring micro-architectures in object-oriented class diagrams. Once SGFinder has detected instances of micro-architectures, we exploit these instances to identify their desirable properties, such as stability, or unwanted properties, such as change or fault proneness. We perform a feasibility study of our approach by applying SGFinder on the reverse-engineered class diagrams of several releases of two Java applications: ArgoUML and Rhino. We characterize and highlight some of the most interesting micro-architectures, *e.g.*, the most fault prone and the most stable, and conclude that SGFinder opens the way to further interesting studies.**

***Keywords*-Micro-architectures, software changes and faults, software maintenance and evolution.**

I. Introduction

Maintenance and evolution are unavoidable activities impacting any long-lived software application. Developers, prior to any such activities, must be aware of previous design and implementation choices to modify the software application appropriately. Design choices include all decisions made by developers when designing and implementing the application. In an object-oriented (OO) application, design choices include the structure of classes and the relations among them. For example, *design patterns* [1] are recurring inter-class patterns that describe solutions to common, recurring design problems in the organization of classes; they are intended to make a design more flexible, reusable, and robust, and thus improve software quality.

On the contrary, *design anti-patterns* [2] have been identified as embodying poor design choices. They are opposite to design patterns [1], *i.e.*, they identify "poor" solutions to recurring design problems, for example Brown's 40 antipatterns describe the most common pitfalls in the software industry [2]. They negatively impact systems by making classes more change-prone and–or fault-prone.

However, only few instances of micro-architectures are usually documented in OO applications, most stemming from implicit design choices. Indeed, there may be many reasons explaining why a micro-architecture is not documented in a catalog. It may happen that it is domain or application specific or that it is the unintended result of several years of evolution leading to the presence of many of its instances. We conjecture that these unknown and unplanned micro-architectures may sometimes have useful properties, such as high stability (*i.e.*, low changing rate) or high fault proneness.

Almost all existing approaches dealing with OO patterns or anti-patterns, and more generally with any micro-architectures, rely on a library of known *abstract* micro-architectures (*e.g.*, design motifs [3]) and a matching algorithm aimed at retrieving those cataloged structures. Micro-architectures have been described as plans and cliches [4], as sets of rules expressed in a domain specific language [5], or as a mix of lightweight code analysis and rules [6]. Few approaches attempted to infer new recurring micro-architectures [7].

We present a novel approach and a tool, SGFinder (Sub Graph Finder) for the inference of OO micro-architectures from class diagrams. In essence, we model a class diagram as a directed graph and define a micro-architecture as the connected [1] subgraph induced by a given subset of classes. Consequently, our tool SGFinder includes a graph representation of the class diagram and an effective and efficient method of enumeration of induced sub-graphs of a given size (*i.e.*, number of nodes).

We apply SGFinder to detect micro-architectures of order three, four, and five on several releases of two Java OO applications: the JavaScript/ECMAScript interpreter Rhino and the UML CASE tool ArgoUML. We then report details on the number and frequency

[1]It is required that there is a chain between any two classes of the subset.

of the detected micro-architectures and summarize findings on their change and fault proneness. We conclude that some micro-architectures have indeed interesting properties.

This paper is organized as follows: Section II summarizes related work; Section III describes the proposed approach to inferring micro-architectures and introduces our sub-graph mining technique along with details of our enumeration strategy. Section IV presents the objects, research questions, and methodology of our case study while Section V discusses its results on several releases of Rhino and ArgoUML. Finally, Section VI concludes with future work.

II. Related Work

Several approaches have been proposed to identify micro-architectures similar to design patterns and anti-patterns. In general, these approaches rely on a library of known motifs and use architectural recovery techniques based on sub-graph matching or on motifs properties.

Rich and Waters [8] proposed the use of constraint programming to recognize plans in Cobol source code. Cobol systems are modeled by their abstract syntax trees. A plan is modeled as nodes of the abstract syntax tree and constraints among nodes (*e.g.*, control- and data-flow). Identification of a plan in source code is converted to a constraint satisfaction problem in which nodes of the plan are variables, relations among nodes are constraints among variables, and the source code abstract syntax tree is the domain of the variables. Recently, Guéhéneuc and Antoniol [3] described an explanation-based constraint programming approach to detect micro-architectures similar to design motifs while taking into account variants semi-automatically.

Other approaches to design pattern identification used clichés recognition algorithms, such as unification, for example the pioneering work by and Krämer and Prechelt [9]. An example is the SOUL environment [10], a logic programming environment based on Smalltalk that directly manipulates Smalltalk constructs through predicates describing the micro-architectures.

Yet, other approaches introduced the use of queries to identify entities whose structure and organization are similar to design motifs [11], [12]. In particular, Keller [12] introduced the SPOOL environment for reverse engineering, which allows manual, semi-automated, or automated identification of abstract design components using queries on source code models.

Generic fuzzy reasoning nets have also been applied to the identification of design motifs [13], [14]. A design motif is described as a generic fuzzy reasoning net representing rules to identify micro-architectures similar to its implementation in source code.

Graphs and graph-transformation techniques also have been used to describe and identify design motifs in system source code [15]. A design motif is described as a graph whose nodes represent entities (classes, interfaces) and whose edges represent relationships among entities. The identification of micro-architectures corresponds to a sub-graph isomorphism: the identification of a sub-graph similar to a given graph in a graph, which is a *difficult* problem [16].

Pettersson and Löwe [17] proposed to transform graphs into planar graphs to improve performance with interesting results. An approach based on graphs and similarity scoring has also been proposed [18].

All the previous works assume the presence of some library of design patterns, design motifs, plans, or micro-architectures. SGFinder does not have a catalog or a set of rules to detect instances of some micro-architectures. Thus, SGFinder is similar to Tonella and Antoniol's previous work [7], in which concept analysis was used to infer domain-specific design patterns. Yet, SGFinder eliminates the problem of manually inspecting concept lattice by relying on the notion of frequent sub-graphs. SGFinder also improves on scalability via an efficient sub-graph enumeration technique.

We draw inspiration from graph and sub-graph mining algorithms used to model and study social or biological networks. In particular, the sub-graph discovery problem divides in two sub-problems: discovering all recurring sub-graphs, as SGFinder does, or only the most frequent ones. Yet, previous works addressing these two sub-problems have limitations: some are restricted to undirected graphs [19], [20], to sub-graphs of less than five nodes [21], to specific sub-graphs [22], or to sub-graphs with limited number of labels on their arcs [23].

III. Our Approach: SGFinder

We now introduce the key concepts to formalize the sub-graph matching problem and describe our algorithm and tool, SGFinder.

A. Notation

We model a class diagram as a labeled graph, with nodes being classes (and interfaces) and arcs representing the relations among classes. A label on an arc linking two nodes specifies the type of relation between two classes (*e.g.*, association, aggregation, or inheritance). For example, Figure 1 displays the labeled graph representing part of the Rhino application. In Figure 1, as well as in all other reported figures, we denote association by 1, aggregation by 2, and inheritance by 3.

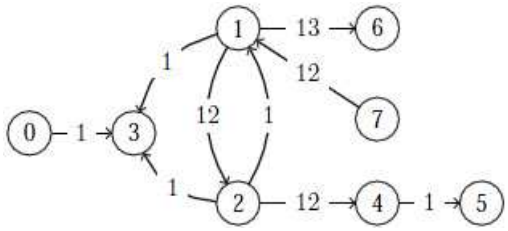

Figure 1. A sub-graph $G(V, E)$ extracted from Rhino (release 1.7R1). Classes are represented by nodes and relations between classes are represented by arcs.

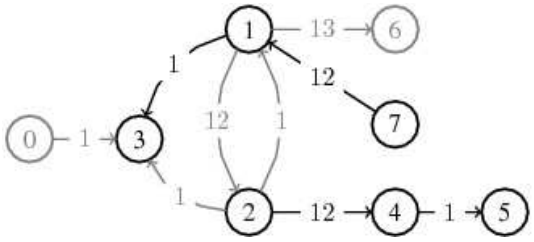

Figure 2. A sub-graph $G(V, E)$ extracted from Rhino (release 1.7R1) with two embeddings of size three $\{7, 1, 3\}$ and $\{2, 4, 5\}$.

Given a set of labels L, a labeled graph is defined by a triplet $G = (V, A, l)$ where V represents the vertex set of G, $A \subseteq V \times V$ the arc set, and $l : A \rightarrow L$ the labeling function, *i.e.*, $l(x, y)$ represents the label of a given arc $(x, y) \in A$. A sequence $\mu = (v_0, \ldots, v_p)$, such that $(v_i, v_{i+1}) \in A$ or $(v_{i+1}, v_i) \in A$ for every $i = 0...p-1$, is named a chain between v_0 and v_p. The length of this chain is p. The distance $dist(x, y)$ between two nodes x and y is the length of the shortest chain between x and y, or ∞ if there is no chain between x and y. A graph is said to be (weakly) connected if there is a chain between any two vertices.

Two graphs $G_1 = (V_1, A_1, l_1)$ and $G_2 = (V_2, A_2, l_2)$ are isomorphic if there exists a one-to-one mapping $\phi : V_1 \rightarrow V_2$ such that, for any x, $y \in V_1$, we have: (1) $(x, y) \in V_1 \Leftrightarrow (\phi(x), \phi(y)) \in V_2$, and (2) $(x, y) \in V_1 \Rightarrow l_1(x, y) = l_2(\phi(x), \phi(y))$. The canonical labeling [24] $cl(G)$ of a graph G is a string of labels with the following property: for any two graphs G_1 and G_2, $cl(G_1) = cl(G_2)$ if and only if G_1 and G_2 are isomorphic. Therefore, we can determine if two graphs are isomorphic by computing and comparing their canonical labellings. (The canonical labeling of a graph can be computed with tools such as the Mckay's Nauty software [25].)

Given a subset $X \subseteq V$ of vertices, the sub-graph of G induced by X, denoted G_X, is the graph which vertex set is X and which arc set contains all the arcs of G that link two vertices of X. An embedding of a sub-graph H of G is a set X of vertices such that G_X is isomorphic to H. For example, in Figure 2, $\{7, 1, 3\}$ and $\{2, 4, 5\}$ are two embeddings of a same sub-graph.

B. Algorithm

Our algorithm takes as inputs a graph G and the size k of the sought sub-graphs (in number of nodes) to be identified and returns, for each different subgraph, its embeddings.

The principle of our algorithm is to generate all k-subsets X of V such that G_X is connected. For each such subset X, our algorithm computes the canonical labeling c of G_X and inserts it into a multi-set M. If c is not already present in M, it means that G_X represents a new sub-graph that has not been discovered before. In this case, the set X is stored into a set S (or simply printed into a file), along with its canonical labeling.

Algorithm 1 SGFinder() procedure.

```
   procedure SGFinder(graph G = (V, A, l), integer k)
2: Choose a vertex x ∈ V
   for i:=0..k − 1 do
4:    Compute L_i = {y ∈ V : dist(x, y) = i}
   end for
6: CompleteSG({}, 0, k, (L_0..L_{k−1}), G)
   SGFinder(G_{V−{x}}, k)
8: end procedure
```

Algorithm 2 CompleteSG() procedure.

```
   procedure CompleteSG( X, j, k, (L_0..L_{k−1}), G )
   if |X| = k then
3:    Build graph H := G_X
      if Connected(H) then
         if c ∉ M then
6:          Insert (c, X) into S
         end if
         Insert c into M
9:    end if
   else
      for all Y ⊆ L_i such that |Y| ≥ 1 and |X| +
      |Y| ≤ k do
12:      CompleteSG(X ∪ Y, j + 1, k, (L_0..L_{k−1}), G)
      end for
   end if
15: end procedure
```

It would be inefficient and time-consuming to generate all k-subsets of vertices before checking for each subset if the induced sub-graph is connected. Instead, our algorithm generates only a limited number of k-subsets of V. The underlying idea is that, if G_X is a connected induced sub-graph of G, then X can not contain two vertices x and y such that $dist(x, y) \geq k$.

This idea is at the core of our efficient enumeration technique, because our algorithm builds (and stores) only useful sub-graphs.

The SGFinder tool implements our algorithm with two procedures named SGFinder() and CompleteSG() shown in Algorithms 1 and 2. A first vertex x is chosen in V (line 2 in procedure SGFinder()). Then, we build k disjoint subsets of vertices, denoted by $L_0 ... L_{k-1}$ (line 3-5), where each L_i contains the vertices which distance to x equals i ($L_0 = \{x\}$). Then, to build X (line 6, developed in the CompleteSG() procedure), we choose the unique vertex present in L_0 (namely x), plus one or more vertices chosen in L_1, plus one or more vertices chosen in L_2, and so on until L_{k-1} (we reach a total number k of vertices). For each set X, we check if G_X is connected, on line 6 in CompleteSG(). At this point of the procedure, we have generated all sub-graphs that contain vertex x. Then, we remove x from the graph and do the same recursively on the residual graph (line 7 in SGFinder()) to build all the other sub-graphs.

Let us consider again Figure 1, let us choose $k = 4$, and let us assume that the first chosen vertex is $x = 0$. We have $L_0 = \{0\}$, $L_1 = \{3\}$, $L_2 = \{1, 2\}$, and $L_3 = \{4, 6, 7\}$. There are now seven possibilities for X: $\{0, 3, 1, 2\}$, $\{0, 3, 1, 4\}$, $\{0, 3, 1, 6\}$, $\{0, 3, 1, 7\}$, $\{0, 3, 2, 4\}$, $\{0, 3, 2, 6\}$, $\{0, 3, 2, 7\}$. (Some of the induced sub-graphs are not connected, *e.g.*, $G_{\{0,3,1,4\}}$ and are thus discarded.)

IV. Case Study

The *goal* of this empirical study is to identify, in OO systems, micro-architectures with desirable or harmful properties. The *quality focus* is to verify applicability of SGFinder to small and medium size OO programs to detect micro-architectures that are stable or fault-prone. The *perspective* is that of both researchers, developers, and managers, who want to get information about undocumented micro-architectures found in OO systems and possible related drawbacks. The *context* of this study are two open-source systems: the Rhino JavaScript/ECMAScript interpreter, and the ArgoUML CASE tool.

A. Objects

We selected Rhino and ArgoUML as systems for our case study because defect and change data are available either from previous authors [26] or from a customized Bugzilla repository for ArgoUML. Table I provides summary data about releases, defects and changes for the two systems.

Rhino[2], the smallest system, is a JavaScript/ECMAScript interpreter and compiler that implements the ECMAScript international standard, ECMA-262 v3 [27]. We downloaded 8 Rhino releases between 1.4R3 to 1.6R1 from the Rhino Web site [3]. Defects were retrieved using data from [26] and numbers of changes were mined from the Rhino CVS logs.

ArgoUML is a UML CASE tool to design and reverse-engineer various kinds of UML diagrams. It is also able to generate source code from diagrams to ease the development of systems. ArgoUML is written in Java. We use 8 development releases from 0.10 to 0.17.5. We extract defect data from the ArgoUML customized Bugzilla repository, *i.e.*, we use the bug-tracking issues identified by the special tag "DEFECT". We then match the bug IDs of the bug tracking issues with the SVN commit messages, as retrieved from the ArgoUML SVN server. Once the file release matching the bug ID is retrieved, we perform a context diff with the previous file release to assign the defect to the appropriate class. We similarly extract change data from the SVN logs.

We recovered the class diagrams of the releases of the systems using the Ptidej tool suite and its PADL meta-model. PADL is a language-independent meta-model to describe the static part and part of the behavior of object-oriented systems similarly to UML class diagrams [3]. It includes a Java parser and a dedicated graph exporter. Relations considered in this study are: (i) associations, (ii) aggregations and (iii) inheritances. It is worth mentioning that mapping class diagrams into graphs generates multi-graphs as it is possible to have more than one kind of relation between two classes.

We applied SGFinder on each release of both systems and extracted micro-architectures of 3, 4 and 5 nodes contained in their class diagrams.

B. Research Questions

We aim at answering the following three research questions:

- **RQ1 – Does SGFinder scale up to medium size programs and what kind of micro-architectures are found in OO systems?** Given the different possible relations between classes in a class diagram, the theoretical combinations for different micro-architectures are enormous and thus it is first necessary to verify that SGFinder can be effectively applied, plus it is of interest to get insight about micro-architectures commonly found in OO systems.
- **RQ2 – Are there micro-architectures particularly fault-prone or fault-free?** Our second RQ is devoted to investigating whether there are some micro-architectures particularly fault-prone or fault-free.

[2]http://www.mozilla.org/rhino/

[3]We targeted subsequent releases with at least ten defects

Systems	Releases (Number Thereof)	Number of			
		Classes	LOCs	Defects	Changes
Rhino	1.4R3–1.6R1 (8)	99–194	21K–75K	12–114	217–1476
ArgoUML	0.10–0.17.5 (8)	876–1243	86K–123K	102–664	541–5048

Table I
SUMMARY OF THE OBJECT SYSTEMS

- **RQ3 – Are there micro-architectures particularly stable or change-prone?** Similarly to RQ2, we analyze the change proneness of the micro-architectures. Empirical and a priori information about the changeability of a micro-architecture could help designers, developers and maintainers.
  Identifying fault or change-prone micro-architectures could prove valuable in conception, development and maintenance tasks once qualitative analysis support the conjecture that it is the micro-architecture structure (classes and relations) responsible for such an unwanted behavior. The qualitative analysis requires a deep understanding of the project history, a thorough analysis of program evolution and the interaction with developers and, for these reasons, it is left as part of the future work. In essence, in this preliminary work we aim at identifying micro-architectures likely responsible for unwanted characteristics but we do not seek to prove any causal relation.

C. Analysis Method

In the following, we first present the information associated to micro-architectures and then how we use such information to answer our research questions.

1) Characterizing the micro-architectures: The first set of features we use to characterize a micro-architecture contains information about its connectivity. To keep things simple, we only consider in our analysis [4] the total number of relations ($nbRel$)[5], the number of associations ($nbAssoc$), the number of aggregations or compositions ($nbAggr$), the number of inheritances ($nbInher$), and the number of "cyclic relations" ($nbCycl$). Cycles are typically generated by pairs of associations, often due to delegation via method calls, *i.e.*, a class A calls or uses a class B and vice-versa.

The second set of information is related to the presence and repartition of the micro-architectures in the studied systems. First we use a variable ($nbReleases$) to count the number of releases in which the considered micro-architecture can be found. For a given release, *i.e.*, in a given class diagram, there are two ways to quantify the presence of a micro-architecture mA_i. The most obvious one consists in incrementing a variable ($nbEmbeddings$) each time our algorithm finds a sub-graph isomorphic to mA_i. However, due to the high connectivity of some nodes, counting frequencies in this way can produce some extremely high and meaningless numbers. To circumvent this, following recommendations in sub-graph mining literature, we impose that newly discovered embeddings of a micro-architecture are counted only when they have no common edges with any of the previously found embeddings. In this way, we only consider distinct graph regions referred to as zones and we report for each micro-architecture its number of zones ($nbZones$). Finally, we consider the set of classes appearing at least once in mA_i and report its cardinality ($nbClasses$)

2) Answering the Research Questions: To answer research questions we resort on descriptive statistics and more precisely on quartiles augmented with minimum and maximum computed over the set of features - presented above - to characterize the micro-architectures. These descriptive statistics are often cumulatively referred to as the five-number summary statistic. For RQ1, we characterize micro-architectures found in Rhino, ArgoUML or both, plus we recorded execution times. To answer RQ2 and RQ3, we consider for each micro-architecture, its set of classes and compute the percentage of its classes that are fault-prone (i.e., with one or more documented fault) or changed classes.

More precisely, as our goal is to locate faulty(change)- prone micro-architectures we adapted precision and recall definition as follows. Consider, fault-proneness; for a given micro-architecture, precision is defined as the ratio of micro-architectures faulty classes over the microarchitecture size averaged over all micro-architecture instances. Thus a 100 % precision means all classes were documented being faulty, while 0 % means fault-free. On the other hand, for a given micro-architecture, recall is computed as the number of classes being faulty and participating in one instance of the micro-architecture over the number of classes in the class diagram.

For a fixed number of classes (e.g., four), we then considered the precision (i.e., average of the number of faults (changes) over the number of its nodes) and use this index to rank micro-architectures from the fault(change)-prone to the fault(change)-free. We then

[4]More connectivity measures were investigated but are not presented in this paper due both to space issues and their weak relevance in RQ2 or RQ3.

[5]Loops are not counted.

	three-nodes (373)	four-nodes (9203)	five-nodes (190061)
nbRel	2,4,5,6,11	3,6,7,9,20	4,8,10,11,27
nbAssoc	0,2,3,4,6	0,4,5,6,12	0,6,7,9,18
nbAggr	0,0,1,2,4	0,1,1,2,7	0,1,1,2,9
nbInher	0,0,1,1,3	0,0,1,2,5	0,0,1,2,6
nbCycl	0,0,1,1,3	0,0,1,2,6	0,0,1,2,8
nbZones	1,1,1,2,68	1,1,1,1,33	1,1,1,1,20
nbClasses	3,3,5,10,140	4,4,5,9,147	5,5,6,10,156
nbReleases	1,2,5,8,8	1,1,3,6,8	1,1,2,4,8

Table II
MICRO-ARCHITECTURES IN RHINO; EACH LINE REPORTS THE FIVE-NUMBER SUMMARY: MIN, Q1, MEDIAN, Q3, MAX

	three-nodes (349)	four-nodes (8224)	five-nodes (180295)
nbRel	2,4,5,6,10	3,6,7,8,17	4,8,9,10,23
nbAssoc	0,2,3,4,6	0,3,4,6,11	0,5,6,8,15
nbAggr	0,0,1,2,4	0,1,1,2,6	0,1,2,2,8
nbInher	0,0,1,1,3	0,0,1,1,5	0,0,1,2,7
nbCycl	0,0,1,1,3	0,0,1,1,5	0,0,1,2,7
nbZones	1,1,1,3,583	1,1,1,2,315	1,1,1,1,171
nbClasses	3,3,6,19,944	4,4,7,16,940	5,5,8,19,944
nbReleases	1,4,7,8,8	1,2,4,7,8	1,1,3,5,8

Table III
MICRO-ARCHITECTURES IN ARGO; EACH LINE REPORTS THE FIVE-NUMBER SUMMARY: MIN, Q1, MEDIAN, Q3, MAX

inspected the top 10% and bottom 10%, of ranked microarchitectures, seeking for hints of what makes those micro-architectures outstanding.

V. RESULTS

In this section, we present results and answer our three research questions, aiming at verifying applicability and obtaining hints on possible usefulness in identifying change or fault-prone classes.

A. RQ1: SGFinder Applicability and Description of the micro-architectures found.

Tables II, III and IV present summary data about the micro-architectures found respectively in Rhino, Argo or both systems. Numbers of different micro-architectures of three, four, or five nodes are indicated

	3-nodes (250)	4-nodes (3993)	5-nodes (52862)
nbRel	2,4,4,5,8	3,5,6,7,11	4,7,8,9,15
nbAssoc	0,2,3,4,6	0,3,4,5,10	0,5,6,7,14
nbAggr	0,0,1,2,4	0,0,1,2,4	0,0,1,2,6
nbInher	0,0,1,1,3	0,0,1,1,4	0,0,1,1,6
nbCycl	0,0,0,1,3	0,0,1,1,4	0,0,1,1,5
nbZones	1,1,2,5,326	1,1,1,2,174	1,1,1,2,95
nbClasses	3,5,8,26,542	4,6,11,24,544	5,8,14,31,540
nbReleases	1,5,7,8,8	1,3,5,7,8	1,3,4,6,8

Table IV
MICRO-ARCHITECTURES IN BOTH RHINO AND ARGO; EACH LINE REPORTS THE FIVE-NUMBER SUMMARY: MIN, Q1, MEDIAN, Q3 AND MAX

in the headers. Five-number summary are provided for each feature in the format Min,Q1,Median,Q2,Max.

Considering the three basic relations (association, aggregation and inheritance), the four derived mixed cases (*e.g.*, two classes can be linked by both aggregation and inheritance, or aggregation and association and so on), and the absence of relations, there are eight possible connections between two graphs. This means that if we consider $n \times n$ pairs of nodes (including loops), we can have, if we do not take into account symmetry, at most $8^{(n\times n-(n-1))} \times 7^{n-1}$ connected subgraphs of n nodes. For instance, one can get about 22×10^{21} different micro-architectures of five nodes. If we consider the union of five-nodes micro-architectures from Rhino and Argo, we only get about 32×10^4 different micro-architectures; this seems to be a very high number but it is just a fraction of the overall number of possible combinations.

Computation times of SGFinder on the studied systems range from a few seconds to two days and half. Extracting 3-nodes micro-architectures takes 20 seconds at most while 4-nodes micro-architectures require less than 30 minutes. The longest computation times occur while extracting five-nodes micro-architectures on the biggest ArgoUML releases we used for this study. In particular, the algorithm took more than 60 hours to retrieve the 82,877 different 5-nodes micro-architectures in ArgoUML0.17.5 and their 13,741,073,588 embeddings. The computation times depended mostly on the edge density of the considered class diagrams. More specifically, the single most time-costly factor is the presence of highly connected nodes which lead to a near-combinatorial explosion of the number of embeddings. Nevertheless, SGFinder was able to retrieve all the micro-architectures (up to 5 nodes) present in the studied releases and we can thus answer positively to the applicability research question sub-part.

As for the number of different micro-architectures, we notice on Tables II and III that it is not strictly linked to the class diagrams' size. Rhino, despite having almost ten times less classes than Argo, presents more combinations of connected graphs for all sizes of micro-architectures. Also, as shown in Table IV, there are many class organizations common to both Rhino and Argo as well as many others (more numerous) specific to each system.

With respect to the connectivity information, the micro-architectures in Rhino seem to have slightly more relations than their counterparts in Argo. Looking at the maximum number of relations, we can notice that some very dense micro-architectures can be found in both OO systems. Figure 3 presents the five-nodes micro-architectures with the highest number of relations in Rhino (27), Argo (23) and their

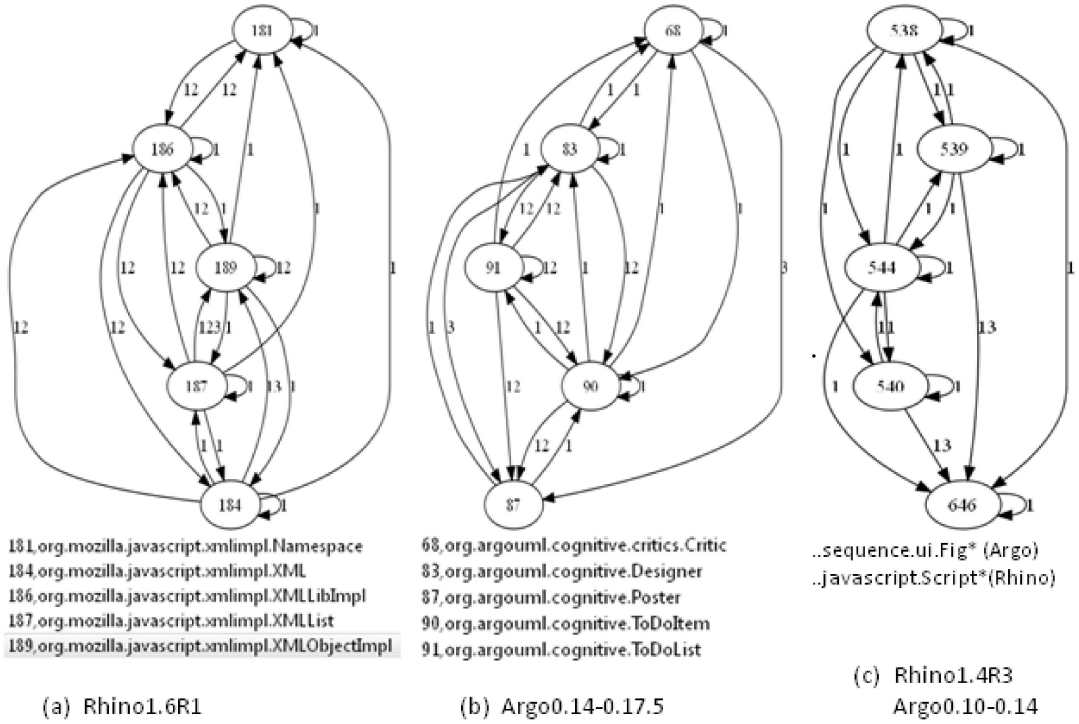

Figure 3. Most Connected 5-nodes micro-architectures

intersection (15).

B. RQ2: Fault-proneness of micro-architectures

Table V is based on micro-architectures present in both Rhino and Argo and report characteristics [6] of the top 10% most faulty (using the precision measure) micro-architectures, as well as the bottom 10% (least faulty micro-architectures). While there are some interesting micro-architectures present either in Rhino or Argo separately, we chose to focus on micro-architectures present in both systems in order to improve the odds of generalization of our findings. Table V support the presence of some micro-architectures particularly fault-prone, as well as other class organizations consistently fault-free. As we analyzed eight Rhino and ArgoUML releases, it is clear that there are micro-architectures constantly present in all Rhino and ArgoUML realses ($nbReleases$ in Table V) and always fault-prone.

Considering five-nodes micro-architectures, precision is as high as 84%; plus the F1 measure is as high as 54%. In other words, some micro-architectures are consistently faulty and/or contain the majority of faulty classes for both systems. At the same time, there are also micro-architectures consistently fault-free (0% precision).

Analyzing connectivity information, we can notice that the most faulty micro-architectures tend to be more connected than the least faulty ones, in particular, they generally contain more associations. Actually, both Spearman and Pearson correlations give values above 0.4 with a confidence level of more than 99% when one evaluates the correlation of the number of associations in the micro-architecture to the precision computed as described in the previous section (Similar correlation values are obtained with cyclic relations). The inverse seems to be true for aggregations and inheritances which tend to be less numerous in most faulty micro-architectures. However, correlation values were only about -0.15.

The information about the presence and repartition of the most or least faulty micro-architectures indicate that most of them are not very common or well spread (generally only one zone), but the maximal values indicate notable exceptions up to seven. This is partially explained by the low percentage of fault-prone classes in both systems. Considering the relatively small numbers of bugged classes in Rhino and Argo the obtained precision numbers are very promising. In particular, we highlight the micro-architecture presented in Figure 4 as an example of a particularly faulty micro-architecture. This structure, present in Rhino (1.5R4, 1.5R4.1) and ArgoUML (0.14-0.16.1), displays an average precision value of 81.25. It represents a category of organizations where (almost) every class "'talks"' to every other class. In fact, when we retrieve the set of micro-architectures

[6]Numbers presented are averages over the two systems.

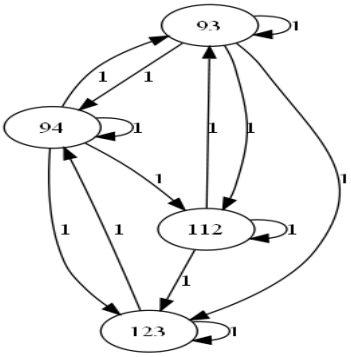

Figure 4. Example of a particularly faulty (precision=81) 4-nodes micro-architecture

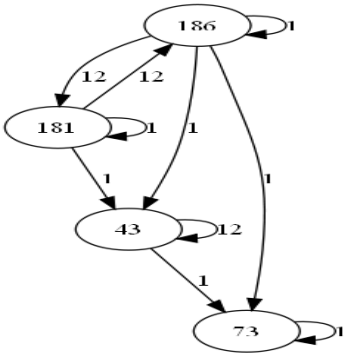

Figure 5. Example of a particularly stable (precision=30) 4-nodes micro-architecture

	three-nodes (250)	four-nodes (3994)	five-nodes (52862)
Precision			
—top10%	41,44,47,52,67	42,45,48,54,83	41,43,46,52,84
—bot10%	0, 5, 9,11,13	0, 8,10,13,14	0, 2, 5, 7,22
F1			
—top10%	5,8,13,17,30	2,10,12,15,40	4,11,14,18,54
—bot10%	0,1, 1, 2, 5	0, 1, 3, 5,16	0, 2, 5, 7,22
nbRel			
—top10%	3,4,6,6,8	4,7,8,8,10	4,8,9,10,14
—bot10%	2,4,4,5,6	3,6,6,7,09	4,7,8,09,13
nbAssoc			
—top10%	2,3,4,5,6	2,5,6,7,10	2,7,8,9,14
—bot10%	0,2,2,3,4	0,3,4,5,08	0,4,5,6,11
nbAggr			
—top10%	0,1,1,2,4	4,7,8,8,10	4,8,9,10,14
—bot10%	0,1,1,2,2	3,6,6,7,09	4,7,8,09,13
nbInher			
—top10%	0,0,0,1,2	0,0,0,1,3	0,0,0,1,4
—bot10%	0,1,1,2,2	0,0,1,1,3	0,0,1,2,5
nbCycl			
—top10%	0,1,1,2,3	0,1,1,2,4	0,1,2,2,5
—bot10%	0,0,0,1,1	0,0,0,1,4	0,0,0,1,5
nbZones			
—top10%	1,1,2,2,9	1,1,1,1,6	1,1,1,1, 7
—bot10%	1,1,1,2,5	1,1,1,1,7	1,1,1,1,14
nbClasses			
—top10%	3,4,5,7,33	4,5,6, 9, 59	1,1,1,1, 7
—bot10%	3,4,5,7,18	4,5,7,12,133	1,1,1,1,14
nbReleases			
—top10%	1,4,5,6,8	1,3,4,5,8	1,2,3,5,8
—bot10%	1,3,4,5,8	1,2,3,5,8	1,2,3,4,8

Table V
MOST AND LEAST FAULTY MICRO-ARCHITECTURES PRESENT IN BOTH RHINO AND ARGO; EACH LINE REPORTS THE FIVE-NUMBER SUMMARY: MIN, Q1, MEDIAN, Q3 AND MAX

with three cyclic relations and where every class is connected to every other class, we obtain a 5-number summary (28,44,56,67,81) with values mostly higher to those from the top 10% most faulty micro-architectures (42,45,48,54,83).

Overall, we can answer positively to the research question as it was possible to find micro-architectures highly fault prone and responsible for a large fraction of faults in both systems.

C. RQ3: Change-proneness of micro-architectures

Similarly to **RQ2**, Table VI reports data about the top 10% most changed and the bottom 10% least changed micro-architectures. Again, we focus on micro-architectures present in both systems. Data reported provides evidence that there are indeed some micro-architecture particularly stable (as few as 10% of changed classes). However, given that changes were very common in the studied systems, our data (with numbers as high as 100%) is less conclusive for most frequently changed micro-architectures.

The analysis of connectivity information points to findings similar to those of **RQ2**: the most changed micro-architectures also tend to have more associations, less aggregations and inheritances.

Considering the high numbers of changed classes in Rhino and Argo some micro-architectures with low precision numbers are worth investigating. In particular, we highlight the micro-architecture presented in Figure 5 as an example of a particularly stable micro-architecture. Present in Rhino (1.6R1) and ArgoUML (0.17.5), this micro-architecture has an average precision value of 30. It presents a ”cascade”-like pattern and we were able to retrieve many other similar organizations with mostly low changeability.

D. Threats to Validity

We demonstrated, in previous sections, the applicability of SGFinder to small-to-medium size programs. Scalability to large programs such as Eclipse or bigger micro-architectures may be problematic and require partitioning large class diagrams into components (e.g subsystems) or the adoption of pruning heuristics.

Threats to *construct validity* concern the relation between the theory and the observation. Here, this threat is mainly due to the need of a manual validation and qualitative analysis. Indeed, we cannot claim any causation effect nor provide any specific interpretation to fault-prone or change(fault)-free micro-architectures. This problem is somehow related to guessing the developers' intent when they planned to implement the micro-architecture under study and the possible drift over time. We can only conjecture that, given the application domain, the classes, methods,

	three-nodes (250)	four-nodes (3994)	five-nodes (52862)
Precision			
—top10%	83,86,88,91,97	85,88,90,93,100	84,86,89,92,100
—bot10%	20,35,38,39,44	17,38,43,48, 50	10,42,47,50, 53
F1			
—top10%	3,5,6,9,27	4,5,6,8,22	5,8,9,12,37
—bot10%	0,1,1,2,13	0,2,3,5,33	0,3,6,11,55
nbRel			
—top10%	3,5,6,6,7	4,6,7,8,10	4,8,9,10,15
—bot10%	2,4,4,6,7	3,5,6,7,10	4,7,8, 9,13
nbAssoc			
—top10%	1,3,4,5,6	2,5,6,7,10	1,6,8,9,13
—bot10%	0,1,2,3,4	0,3,4,5,08	1,4,5,7,11
nbAggr			
—top10%	0,0,0,1,3	0,0,1,2,4	0,0,1,1,5
—bot10%	0,1,2,2,3	0,1,2,2,4	0,1,2,2,6
nbInher			
—top10%	0,0,1,1,2	0,0,0,1,4	0,0,0,1,5
—bot10%	0,1,1,2,2	0,0,1,1,3	0,0,1,1,4
nbCycl			
—top10%	0,1,1,2,3	0,1,1,2,4	0,0,1,2,5
—bot10%	0,0,1,1,2	0,1,1,2,3	0,0,1,1,4
nbZones			
—top10%	1,1,2,2,5	1,1,1,1, 5	1,1,1,1,10
—bot10%	1,1,1,2,5	1,1,1,1,11	1,1,1,1,10
nbClasses			
—top10%	3,4,5,7,26	4,5,6, 8, 90	5,6, 8,11,158
—bot10%	3,4,5,7,69	4,5,8,14,321	5,8,13,29,404
nbReleases			
—top10%	1,4,5,7,8	1,2,4,5,8	1,2,3,4,8
—bot10%	2,4,5,7,7	1,3,4,5,8	1,2,3,5,8

Table VI
MOST AND LEAST CHANGED MICRO-ARCHITECTURES PRESENT IN BOTH RHINO AND ARGO; EACH LINE REPORTS THE FIVE-NUMBER SUMMARY: MIN, Q1, MEDIAN, Q3 AND MAX

and relations (and the general information that can be extracted from the source code and documentation), it will be possible to infer the developers' *likely* intent and understand the drift toward a micro-architecture with unwanted features.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, these threats can be due to subjectiveness in distinguishing between associations or aggregations and the number of faults assigned to classes. We attempted to avoid any bias by using well consolidated tools, reusing defect data provided by other researchers and extracting facts from source code and bug tracking repositories.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. We do not claim any relation between the micro-architectures and unwanted features and the existence of possible relations is left to the developers' judgment and experience. In essence, SGFinder, reports micro-architectures but no claim is made and, in this preliminary work, we limit ourselves to highlight micro-architectures with high precision and recall with respect to fault-proneness or stability.

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to two systems: Rhino and ArgoUML. Yet, our approach is applicable to any other system of comparable size. However, we cannot claim that similar results would be obtained with other systems and that micro-architectures common across systems and releases will always be detected. However, the two systems correspond to different domains and applications, have different sizes, are developed by different teams. We believe this choice mitigates the threats to the external validity of our study.

VI. CONCLUSION

In OO software, micro-architectures, likewise design patterns, are recurring classes and relations organizations. We presented SGFinder, an algorithm and a tool to support micro-architecture discovery based on a reformulation as a sub-graph mining problem. SGFinder uses an effective enumeration technique that allows us to infer instances of micro-architectures in small to medium size programs and to study micro-architecture properties such as stability or fault proneness. To the best of our knowledge only a few works addresses the same or similar problem [7].

We used SGFinder on eight releases of two well known Java applications: the Rhino JavaScript/ECMAScript interpreter and ArgoUML, a UML CASE tool to design and reverse-engineer various kinds of UML diagrams. After providing insight about the kind of micro-architectures (of three, four or five nodes) found in OO systems, we focused on fault-proneness and changeability. We characterize the most and least faulty/changed micro-architectures and report some of the most interesting micro-architectures with respect to their connectivity and frequency.

Despite the encouraging results, more work is needed to (i) further optimize the proposed algorithm and gain in scalability, (ii) define heuristics and rules able to classify micro-architectures discovered in a system under development, and (iii) go beyond the micro-architectures and analyze, similarly to design patterns, the roles played by participant classes, and (iv) provide qualitative analysis and validation of the findings. For generalization purposes, our plan for future work also include replication of the study on different systems.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

[2] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[3] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008.

[4] V. Kozaczynski, J. Q. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Trans. on Software Engineering*, vol. 18, no. 12, pp. 1065–1075, Dec 1992.

[5] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[6] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," *Automated Software Engineering, International Conference on*, vol. 0, pp. 123–134, 2006.

[7] P. Tonella and G. Antoniol, "Inference of object oriented design patterns," *Journal of Software Maintenance - Research and Practice*, vol. 13, no. 5, pp. 309–330, September-October 2001.

[8] C. Rich and R. C. Waters, *The Programmer's Apprentice*, 1^{st} ed. ACM Press Frontier Series and Addison-Wesley, January 1990.

[9] C. Krämer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, L. M. Wills and I. Baxter, Eds. IEEE Computer Society Press, November 1996, pp. 208–215.

[10] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Proceedings of the* 26^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, J. Gil, Ed. IEEE Computer Society Press, August 1998, pp. 112–124.

[11] B. Kullbach and A. Winter, "Querying as an enabling technology in software reengineering," in *Proceedings of the* 3^{rd} *Conference on Software Maintenance and Reengineering*, P. Nesi and C. Verhoef, Eds. IEEE Computer Society Press, March 1999, pp. 42–50.

[12] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse-engineering of design components," in *Proceedings of the* 21^{st} *International Conference on Software Engineering*, D. Garlan and J. Kramer, Eds. ACM Press, May 1999, pp. 226–235.

[13] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proceedings of the* 24^{th} *International Conference on Software Engineering*, M. Young and J. Magee, Eds. ACM Press, May 2002, pp. 338–348.

[14] J. H. Jahnke and A. Zündorf, "Rewriting poor design patterns by good design patterns," in *Proceedings the* 1^{st} *ESEC/FSE workshop on Object-Oriented Reengineering*, S. Demeyer and H. C. Gall, Eds. Distributed Systems Group, Technical University of Vienna, September 1997.

[15] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of Java software," in *Proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*, B. Scherlis, Ed. ACM Press, November 1998, pp. 10–16.

[16] D. Eppstein, "Subgraph isomorphism in planar graphs and related problems," in *Proceedings of the* 6^{th} *annual Symposium On Discrete Algorithms*, K. Clarkson, Ed. ACM Press, January 1995, pp. 632–640.

[17] N. Pettersson and W. Löwe, "Efficient and accurate software pattern detection," in *Proceedings of the* 13^{th} *Asia Pacific Software Engineering Conference*, P. Jalote, Ed. IEEE Computer Society Press, December 2006, pp. 317–326.

[18] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *Transactions on Software Engineering*, vol. 32, no. 11, November 2006.

[19] J. Chen, W. Hsu, M. Lee, and S. Ng, "Nemofinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs," *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 106–115, 2006.

[20] Z. Razaghi and M. Kashani, "Kavosh: a new algorithm for finding network motifs," *Bioinformatics*, vol. 10, pp. 0–0, 2009.

[21] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[22] V. Batagelj and A. Mrvar, "Pajek-analysis and visualization of large networks," *Springer-Verlag*, no. 2265, pp. 77–103, 2003.

[23] S. Wernicke and F. Rasche, "A tool for fast network motif detection," *Bioinformatics*, vol. 22, pp. 1152–1153, 2006.

[24] J. Huan, W. Wang, and J. Prins, "Efficient mining of frequent subgraphs in the presence of isomorphism," *Third IEEE International Conference on Data Mining (ICDM'03)*, p. 549, 2003.

[25] B. Mckay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[26] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Transaction on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[27] ECMA, *ECMAScript Standard - ECMA-262 v3*. ISO/IEC 16262, 2007.