A Graph Pattern Matching Approach to Software Architecture Recovery [*]

Kamran Sartipi[1] Kostas Kontogiannis[2]

University of Waterloo
Dept. of Computer Science[1] and,
Dept. of Electrical & Computer Engineering[2]
Waterloo, ON. N2L 3G1, Canada
{*ksartipi, kostas*}*@swen.uwaterloo.ca*

Abstract

This paper presents a technique for recovering the high level design of legacy software systems based on pattern matching and user defined architectural patterns. Architectural patterns are represented using a description language that is mapped to an attributed relational graph and allows to specify the legacy system components and their data and control flow interactions. Such pattern descriptions are viewed as queries that are applied against an entity-relation graph that represents information extracted from the source code of the software system. A multi-phase branch and bound search algorithm with a forward checking mechanism controls the matching process of the two graphs by which, the query is satisfied and its variables are instantiated. An association based scoring mechanism is used to rank the alternative results generated by the matching process. Experimental results of applying the technique on the Xfig system are also presented.

1 Introduction

The inherent complexity of large legacy software systems has been regarded as a major issue for the maintenance and evolution of these systems. Due to prolonged maintenance, the architectural design of a legacy system constantly deviates from its original design. In this context, architectural recovery has been considered as the front-line for many software analysis and reengineering activities. Different approaches view the software architecture recovery as a clustering problem [8], constraint satisfaction problem (CSP) [20], graph partitioning problem [4], and visualization and composition problem [11, 6]. We view the architectural recovery as an *approximate graph matching* problem. In this approach, an Architectural Query Language (AQL) provides means for representing high level descriptions of the software system usually referred to as the *conceptual architecture* of the system. An inexact graph matching engine provides an optimal matching between the graph that originates from the AQL query, and the graph that represents the data and control flow properties of the software system obtained from parsing the source code. In this context, the pattern matching can be viewed as a process that determines an optimal sequence of graph edit operations (insertion, deletion, relabeling) among nodes and edges of the two graphs, so that a given query (i.e., architectural description) can be satisfied by information obtained from the source code of the software system.

In real world applications such as: image processing, pattern recognition, circuit layout design, computer network routing, traffic control, and software reverse engineering, an interesting problem is to decompose a large input graph into regions (partitions) with particular topological properties and region inter-connections which conform with a generic pattern. In such applications, the user defines a generic pattern and the desired topological property of the regions. The pattern matching engine then tries to find a sub-graph of the input graph that closely (not exactly) matches with the generic pattern (i.e., *inexact graph matching* [18]).

The proposed graph matching process consists of two major phases: i) restricting the search space of the graph matching by pre-processing the entire input graph and producing a database of graph regions, each loosely satisfying a desired property; and ii) applying a graph matching algorithm that approximately matches an architectural pattern, represented as a query, against the database of graph regions.

[*]This work was funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto) and the National Research Council of Canada.

2 Related work

The following approaches to software architectural recovery use search techniques to recover a defined pattern in a software system. The Murphy's reflexion model allows the user to test a high level conceptual model of the system against the existing high level relations between the system's modules [12]. Kazman evaluates the architectural complexity of a system by searching for architectural patterns [7]. Some clustering techniques also provide modularization of a software system based on file interactions and partitioning methods [8]. In contrast to these works, we generate an abstract pattern graph as a query whose properties can be arbitrary changed, and then an approximation of this pattern is found in the software system graph.

A number of researchers have investigated the application of graph matching in different problem domains. Messmer compares an input graph with a collection of prototype graphs by decomposing the prototypes into primitive graphs which are stored in a database, and comparing them against the primitives of the input graph [9]. Eshera and Fu decompose the matching graphs into simple trees to be matched [5]. Shapiro and Haralick define structural description with weighted nodes and edges to evaluate cost for inexact matching [18]. Bunke and Allermann use graph edit operations and generate a state space to be searched for a minimum path [3]. In our approach, we generate a database of graph regions and incrementally match a pattern graph against this database, which is close to the approaches proposed by Messmer and Eshera.

The technique in this paper also relates to our previous work on developing a software architecture recovery framework. In [16], we presented an Architectural Query Language (AQL) for describing the high-level abstraction of a software system in terms of abstract modules and interconnections, whose module variables are to be instantiated by a search engine. We also applied data mining techniques on the reverse engineering domain. In [17], we presented the software architecture recovery as a Valued Constraint Satisfaction (VCSP) problem using the AQL queries, where the module properties and the links between modules are viewed as constraints among the module variables to be satisfied. In the current paper, the AQL query is presented as an abstract pattern graph which is expanded into a pattern graph and then inexactly matched against a database of graph regions extracted from the software system.

3 A framework for architectural recovery

The proposed framework for software architecture recovery consists of three phases (Figure 1):

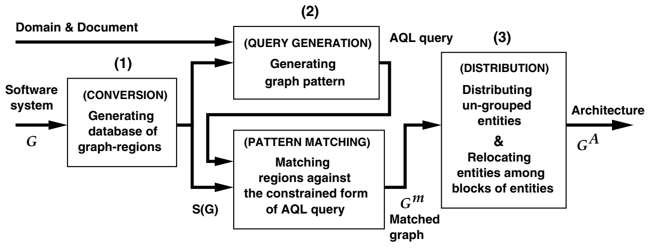


Figure 1. The graph pattern matching framework for software architecture recovery.

In the first phase (*conversion*), the software system is parsed and the source code entities (i.e., *file, function, variable*, and *type*)[1] and their relationships (i.e., *call, define, set, update*, and *declare*), are extracted. The extracted low-level relations are aggregated into more abstract relations (i.e., *call* and *use-resource*) to generate the source model graph G (Figure 2). Based on data mining techniques, graph G is partitioned into a collection of highly connected regions, which serve as the source model database for the matching engine. The application of data mining technique in partitioning process has been discussed in [17].

In the second phase (*analysis*), based on: maintainer's knowledge of domain, system document inspection, and/or source model analysis, an abstract pattern of the system architecture is formulated in the form of an AQL query. The AQL query is expanded into a pattern graph G^p which is approximately matched against the source model graph G. In other words, the pattern matching algorithm results in a graph G^m which, after some insertion, deletion, and relabeling operations, is isomorphic or similar to the pattern graph G^p. This phase presents the contributions of the current paper.

In the third phase (*distribution*), a number of unresolved source model entities can be distributed among the blocks of the recovered architecture, or the entities in the blocks can be selectively moved between the blocks based on: overall closeness between the entities, or user inspection. This phase represents the *user involvement* in the recovery process.

4 An abstract query language

In this section, we briefly describe a formalism for specifying a conceptual architecture denoted as Architectural Query Language (AQL), which represents an architectural graph-pattern using *abstract blocks* and *abstract links*[2] in an AQL query. The AQL includes directives for the search en-

[1] We consider *function*, *type*, and *variable* as *atomic entities* and *file* as *composite entity*, however, we refer to each of them as an "entity".

[2] An abstract block and an abstract link correspond to a group of nodes and a group of edges in G, respectively.

gine (inexact graph matching) to allow the user interaction and control of the architecture recovery process.

Some of the AQL features include: i) gradually increasing the number of abstract blocks and abstract links to allow partial matching; ii) applying hierarchical pattern matching with different granularity for entities, i.e., system level analysis on a collection of files and subsystem level analysis on a collection of functions, types, and variables; iii) defining fixed system entities as *seeds* to be included in the final result; iv) merging two or more abstract blocks of the AQL query into one block in order to reduce the complexity of a part of the pattern; and v) manipulating the result of matching process in order to meet the architectural recovery objectives. A part of an AQL query, consisting of an abstract subsystem *S1* (of files) and abstract links between *S1* and other subsystems, is shown below:

```
BEGIN-AQL
SUBSYSTEM: S1
   MAIN-SEEDS:        file e_edit, e_update, e_flip
   IMPORTS:
     RESOURCES:       rsrc ?IR,
                      rsrc ?R1(1 .. 10) S2,
                      rsrc ?R2(7 .. 20) S4
   EXPORTS:
     RESOURCES:       rsrc ?ER,
                      rsrc ?R3(1 .. 15) S2,
                      rsrc ?R4(1 .. 5) S3
   CONTAINS:
     FILES:           file $CL(3 .. 10),
                      file e_edit, e_update, e_flip
   RELOCATES:      YES:
                      file e_allign, u_scale   TO: S3
END-BLOCK
. . . . .
END AQL
```

The notations *?IR* and *?ER* in the import and export parts denote two unidentified numbers of links between the current block and any other block in the query, where, their interactions have not been constrained by the AQL query. Therefore, *?IR* and *?ER* are not matched by the matching algorithm, however, their instantiation of links are shown in the result of the analysis (i.e., G^A) to be used for further adjustment of the architectural pattern. "*?Ru*(*x..y*)" represents a constrained abstract link *?Ru* of type *use-resource* with *x* and *y* as the minimum and maximum quantities for expanding *?Ru*. "$CL(*x..y*)" (as *contains file*) represents an abstract block containing the nodes of type $file$ with specified min/max threshold numbers for block expansion.

The above AQL fragment is interpreted as: a subsystem named S1 that definitely contains files *e_edit.c, e_update.c* and *e_flip.c* (*main seeds*), imports minimum one and maximum ten resources (?R1) from subsystem S2. A similar interpretation holds for the *EXPORTS* and *CONTAINS* sections of the query. The abstract graph corresponding to the complete form of the above AQL query will be presented in section 8 (experiments).

5 Architectural graph pattern generation

In the pattern based architectural recovery, patterns are formed based on: i) mapping the source code to a reference architecture to find core system entities for each block of the reference architecture; ii) available system architecture document or consulting with the system developers; iii) analyzing the association property among the system files using a component graph [14]; or iv) clustering technique [15]. The objective in any of these methods is to extract a small group of system entities which represent the core functionality of a block in the system architecture.

The groups of core entities in the abstract blocks are identified as main-seeds for subsystems (or *modules*) in the AQL query. The core entities determine the initial minimum sizes of the abstract nodes and their interactions. However, the minimum and maximum sizes are determined in an iterative pattern matching process based on the user's desire or the result of a previous run of the matching process in order to restructure the system based on high cohesion and low coupling property. In a typical scenario, the user defines the architectural pattern of a system using an AQL query, and tries to restructure the system by constraining the pattern and approximately matching it against the software system. An example of pattern recovery is presented in section 8.

6 Graph based system representation

Attributed Relational Graphs (*ARG*) are frequently used in representing a real world system of objects and relationships. Moreover, they provide a valuable modeling abstraction for graph matching problems [9, 5, 18, 3]. We use attributed relational graphs for representing entities and relationships in a software system.

6.1 Graph definitions

In this section, we summarize the underlying concepts of the *attributed relational graph* used in our work.

Definition 1: Let L be the set of symbolic labels obtained from an alphabet $\mathcal{L}$, A be the set of attribute values obtained from an alphabet $\mathcal{A}$, and k be an integer. We define $L \times A^k$ as the set of all possible label-attributes[3] for nodes and edges. An ARG is defined as a two-tuple $G = (N, R)$,

[3]For simplicity we refer to the term *label-attribute* as *attribute*.

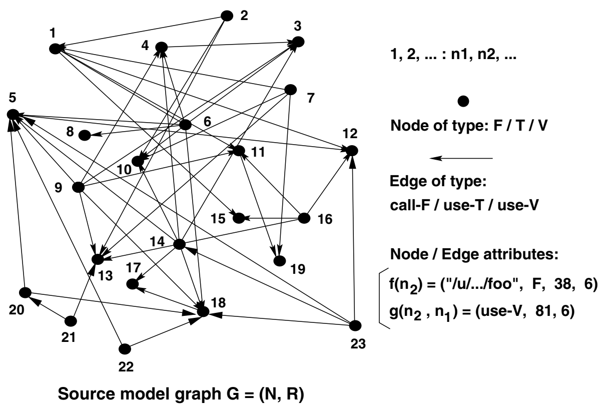
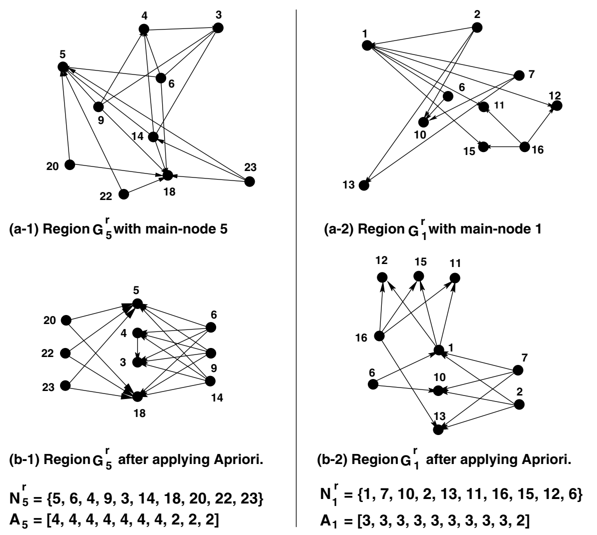


Figure 2. An ARG G of a software system.



(a-1) Region G^r_5 with main-node 5
(a-2) Region G^r_1 with main-node 1
(b-1) Region G^r_5 after applying Apriori.
(b-2) Region G^r_1 after applying Apriori.
N^r_5 = {5, 6, 4, 9, 3, 14, 18, 20, 22, 23}
A_5 = [4, 4, 4, 4, 4, 4, 4, 2, 2, 2]
N^r_1 = {1, 7, 10, 2, 13, 11, 16, 15, 12, 6}
A_1 = [3, 3, 3, 3, 3, 3, 3, 3, 3, 2]

Figure 3. (a) Two regions of graph G in Figure 2 selected using the association property. (b) The Apriori algorithm is used to reveal the associated groups with maximum degrees.

where $N = \{n_1, n_2, ..., n_n\}$ is the set of attributed vertices (*nodes*) and $R = \{r_1, r_2, ..., r_m\}$ is the set of directed attributed edges. Two labeling functions $f : N \to L \times A^k$ and $g : N \times N \to A^k$ return the node and edge attributes, respectively:

$$f = \{(n_i, a) \mid 1 \le i \le n,\ a \in L \times A^k\}, \text{and}$$
$$g = \{(n_i, n_j, e) \mid 1 \le\ i, j\ \le n,\ e \in A^k\}.$$

Where, $(n_i, a) \in f$ represents node n_i with a as its attribute, and $(n_i, n_j, e) \in g$ represents a directed edge from node n_i to node n_j with e as its attribute.

Figure 2 represents the ARG of a software system with 23 nodes where each node or edge has the following attributes:
Label: a string denoting a unique name for each entity in the software system, i.e., a full path name (edges do not have labels);
Type: a specifier that classifies the nodes of a graph into different categories (*file, func, aggregate-type*, and *global-variable*)[4], or classifies the edges into categories (*call-F, use-T*, and *use-V*);
Location: two integers for *file number* and *line number in file*.

The node and edge labeling functions f and g return the attributes of nodes and edges, for example:
$f(n_2)$ = (*"/u/.../foo"*, F, 38, 6) indicates that node n_2 with label *"/u/.../foo"* is a function (type *F*) which has been defined in line 38 of the source file 6; and $g((n_2, n_1))$ = (*use-V, 81, 6)* indicates that the function "foo" references a global variable, represented as node n_1 in line 86 of file 6, whose attributes are returned by $f(n_1)$.

Definition 2: *Association* in a group of graph nodes is a property "A" where two or more source nodes share one or more sink nodes (through direct graph edges). In this sense, the group of source and sink nodes are denoted as an *associated group*. The *association degree* between the nodes of an associated group is the number of sink nodes, and the *association support* is the number of source nodes in that group. In Figure 3(b-1), the group of nodes 5, 6, 4, 9, 3, 14, 18 are associated with association degree 4 (i.e., 4 sink nodes) and associated support 3 (i.e., 3 source nodes); and the group of nodes 5, 20, 18, 22, 23 are associated with degree and support 2 and 3, respectively.

Revealing all the associated groups in a large graph is computationally expensive. We use the Apriori algorithm [2], originally presented in the data mining domain, to extract the groups of nodes with maximum association degrees. A more detailed discussion on the application of the data mining on reverse engineering can be found in [16, 10].

Definition 3: A *region* $G^r_j = (N^r_j, R^r_j)$ of a graph $G = (N, R)$ is a subgraph of G (i.e., $N^r_j \subseteq N$ and $R^r_j \subseteq R$) that corresponds to a node n_j in that region (i.e., $n_j \in N^r_j$). In a region G^r_j each node $n_i \neq n_j$ satisfies the association property "A" with respect to node n_j. We call n_j the *main-node* of region G^r_j.

In general, different regions of a graph have a number of shared nodes. Figure 3(a-1) represents region G^r_5 of the source model graph G that satisfies the association properly "A", i.e., each node is a member of an associated group with respect to node 5. However, it is not clear what is the

[4]In this paper we refer to *aggregate-type* and *global-variable* as *type* and *var*, respectively.

highest association degree of each node in G_r^5 with regard to node 5, since each node can be a member of different associated groups having a different association degree (regarding node 5) in each group. The Apriori algorithm is used to extract all the associated groups in a region, and allows us to determine the maximum association degree of each node with respect to the main-node of that region. Figures 3(b-1) and (b-2) illustrate the application of the Apriori algorithm on the regions in Figures 3(a-1) and (a-2), respectively. The nodes of a region are ranked according to: i) the maximum association degree with the region's main-node; and ii) the constraint of collecting equal number of source and sink nodes.

The *region database* of a graph $G = (N, R)$, denoted as *S(G)*, is a collection of all regions G_j^r of G, along with their ranking in graph G based on the average of maximum association degrees of nodes in each region. In ranking the regions G_5^r and G_1^r (Figure 3) based on their average of maximum association degrees, G_5^r has a higher rank than G_1^r. We call a sub-graph of a region a *sub-region*.

Definition 4: A *pattern graph* $G^p = (N^p, R^p)$ is generated from an AQL query by expanding the abstract blocks and links. Below, a fragment of a simple AQL query with two modules M_1 and M_2 (referred to as abstract blocks ab_5 and ab_1, named after their main-seeds n_5 and n_1) and one abstract link al_1 is shown:

```
MODULE: M1
  MAIN-SEED:        func n5
  EXPORTS:
    FUNCTIONS:      func al1(1..2) M2
  CONTAINS:
    FUNCTIONS:      func $CF(2..4)
END-BLOCK

MODULE: M2
  MAIN-SEED:        func n1
  IMPORTS:
    FUNCTIONS:      func al1(1..2) M1
  CONTAINS:
    FUNCTIONS:      func $CF(2..3)
END-BLOCK
```

The graph representation of this AQL query and its expansion to pattern graph G^p are illustrated in Figure 4. In this example, we only consider the blocks with only one *node type F* (i.e., function). However, the discussion is valid for an AQL query with multiple node types (i.e., function, type, and variable).

In general, an AQL query can generate many pattern graphs based on the integer range (*min, max*) associated with each abstract block and abstract link. We are interested in the *maximal* pattern graph using the maximum values of the ranges.

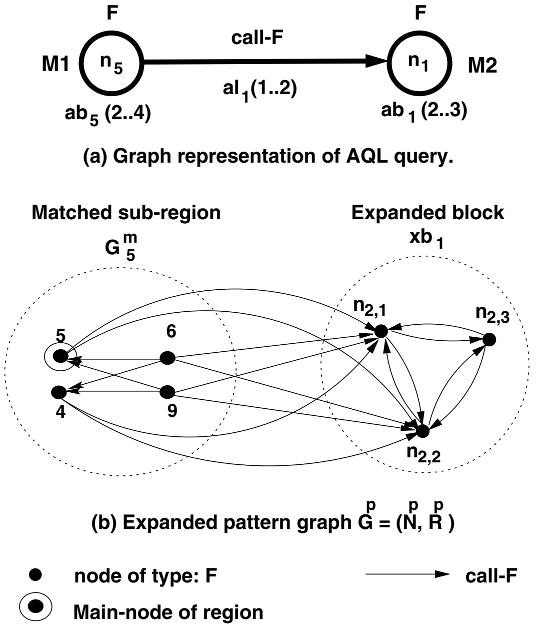


Figure 4. The expansion of an AQL query.

In this example, we consider that the abstract block ab_5 has already been expanded into xb_5 and matched with region G_5^r, and the result is represented as the matched sub-region G_5^m in the first phase of the matching process.

The generation of the maximal pattern graph G^p proceeds as follows:

- For the current abstract block (e.g., ab_1): i) an expanded block xb_1 is generated with maximum number of nodes (i.e., 3 nodes of type function for ab_1), and; ii) edges with label *call-F* connect every node in xb_1 to every other node in xb_1.
- For each abstract link (e.g., $al_k(p, q)$), q groups of edges are generated, each group connect every node from the source "matched sub-region" to one node in the sink "expanded block", or vice versa. Initially, the first q nodes are selected as the sink nodes, however during the matching process the sink node of a group of edges that are not matched yet can be changed to another node without any cost. We denote this operation as "*edge-sink-change*". For example, the abstract link $al_1(1..2)$ represents two groups of edges with label *call-F* between every nodes in G_5^m and the first two nodes in xb_1.

The rational for such a pattern graph is to search for cohesive sub-regions that demonstrate maximal association among their own entities in terms of functions that call each other and functions that all use the same group of variables and data types. Since the chance of finding such groups of entities in a software system is slim, we allow inexact match between the pattern and candidate sub-regions to find the closest sub-regions to this pattern.

6.2 Graph distance

In this section, we outline the concepts pertaining to the distance between two attributed relational graphs.

Definition 5: The *distance* between two ARGs $G1$ and $G2$ (shown as $dist(G1, G2)$) is defined as the minimum cost of a sequence of changes or *graph edit operations* that must be performed on one graph (e.g., $G1$) in order to produce the other graph (i.e., $G2$). These changes are usually in the form of node or edge *deletion, insertion*, or *relabeling* [5].

In this context, we perform the graph edit operations on a "selected region and its edges to the matched sub-regions" to match it with the "current expanded block and its edges to the matched sub-regions", where, we refer to "edges to the matched sub-regions" as *glue edges*. Since we do not match the labels of the nodes between the regions and expanded blocks, hence, no relabeling is performed. We can always select a candidate region with equal or more nodes than the current expanded block, hence, we can also avoid *node insertion*. For the rest of the cases, a certain cost is associated with each graph edit operation that corresponds to matching one node from the selected region with one node from the expanded block. The costs for edge insertion, edge deletion, and node deletion, denoted as c_{ei}, c_{ed}, and c_{nd}, are as follows:

- *edge insertion cost* (c_{ei}): i) for *inter-region* edges (c_{ei}^{out}), if the insertion of edges to the current node does not violate the specified minimum number of edges between the resulting sub-regions then the cost is 1 for the first edge insertion between the sub-regions and zero for the rest of the edge insertions[5]. Otherwise, if the minimum number of edges is violated the cost is top cost; ii) for *intra-region* edges (c_{ei}^{in}), the cost is proportional to the inverse of the association degree between the corresponding nodes of the inserted edge (limited to cost 1 for each insertion).

- *edge deletion cost* (c_{ed}): i) for *inter-region* edges (c_{ed}^{out}), the cost is zero if there is no abstract link between the corresponding abstract blocks (i.e., the deleted edge is not part of the pattern); and the cost is top cost if there are edges between the corresponding abstract blocks[6] (i.e., the existing edge has violated the maximum number of edges between the sub-regions); ii) for *intra-region* edges (c_{ed}^{in}), this cost does not apply since there is no edge in a region whose matching edge inside the current expanded blocks does not exist (blocks are maximally expanded).

- *node deletion cost* (c_{nd}): we assume that the main-seeds and seeds in the AQL query are already excluded from the nodes of the current region. The current node n_d is deleted from the current region *only if* it is the same as a node n_k in a previously matched sub-region and n_k has matched inter-region link(s). In this case, node n_d is deleted with top cost. When the matching process is finished, all remaining nodes of the current region that have not been matched will be deleted with zero cost.

[5]That is, if there is no edges (with desired direction) between the current node in the current region and any other nodes in the related matched sub-region, then the cost is 1, otherwise, if there exist one or more edges between them, then the cost is zero.

[6]Note that the *edge-sink-change* (see definition 4) must be checked to adjust the current expanded block so that edge deletion is performed only if there is no other option.

In the case of *incoming* inter-region edges into the current expanded block, if the number of already matched sink nodes plus the number of yet unmatched potential sink nodes, is equal or greater than the minimum number of corresponding incoming abstract links, then the minimum range condition has not been violated yet. This criterion is the basis for the *forward checking* mechanism to limit the search space. An example of cost calculation is presented in the next section.

6.3 Multi-stage state space representation

As it was shown before, a pattern graph G^p consists of a number of smaller patterns, i.e., the expanded blocks, to be matched against the regions of the input graph G. This characteristic allows us to manage the complexity of the matching process of a large input graph against a region-wise pattern graph. In this form, the whole matching process can be divided into incremental sub-matchings, each taking care of an expanded block, and incrementally generating an approximate matching between the pattern graph and the source graph.

In order to implement an incremental matching, a *multi-stage search-space* is generated during the matching process. Each stage S_i corresponds to a $phase\ i$ ($i : 1, 2, 3, ...$) of the matching process, in which a sub-region of G_j^r is matched against the expanded block xb_j.

Each stage S_i is a *decision tree* whose nodes represent the states of the matching process. In each state, the cost of graph edit operations for matching a candidate node from the region G_j^r with a node from the expanded block xb_j is evaluated. The decision tree consists of: i) a *root node* that matches with the main-node of the region; ii) a number of *internal nodes* at different *levels* of the tree containing the partial matching of the nodes from the region and expanded block; and iii) *leaf nodes* at the lowest level of the tree, containing the full matching.

At each level of the tree, every remaining node (i.e., not matched yet) of the region is checked against *only one* of the remaining nodes of the expanded block xb_j. This is because in xb_j, all *unmatched nodes* with incoming edges are the same and all *unmatched nodes* without incoming edges are also the same. This topological property of xb_j eliminates the need for checking all the remaining nodes of xb_j. Therefore, for each node matching we choose a proper node (with or without incoming edges) from the expanded block. If such a node does not exist then the graph edit operations with associated costs are applied.

The first v levels of the tree ($1 < v \leq u$, u is size of the expanded block) may be allocated to matching v user selected nodes of the region (called *seeds*). In this case, at each level only a seed node is matched. The states of each stage contain a fixed part corresponding to the matched sub-region nodes that are accumulated from the stages S_1 to S_{i-1}.

Figures 6 represents the second stage[7] of the search space corresponding to the matching process shown in Figure 5.

7 Inexact pattern matching

In *exact matching*, one is concerned about finding a subgraph of graph $G1$ that is isomorphic with another graph $G2$, whereas, in *inexact matching* we are interested in identifying the optimal sequence of graph edit operations that can be applied on one graph in order to make parts of the two graphs isomorphic [18, 5, 3]. In most real applications due to the effect of noise, distortion, sampling error, or lack of a known or fixed pattern, the exact matching is not possible. In such situations, finding a subgraph of the input graph that is similar enough to a given pattern graph is interested.

Problem definition:
Given an AQL query aql, an input graph $G = (N, R)$, a region property "A", and a graph distance threshold d_t, find a subgraph of G (called G^m) that inexactly matches with the expansion of aql query (called G^p), so that $dist(G^p, G^m) \leq d_t$.

Algorithm:
The inexact matching process is performed in three steps:

Step 1:
In an off-line process, the input graph $G = (N, R)$ is decomposed into $n = |N|$ regions (each corresponding to a main-node n_j) based on the association property "A". In each region G^r_j, the nodes are ordered based on their association degrees with the main-node n_j. This node ordering causes the matching algorithm to first test the highly associated nodes. The regions of the graph G are also ranked and stored in the region database *S(G)*.

[7]Because of space limitation, the first stage is not discussed here.

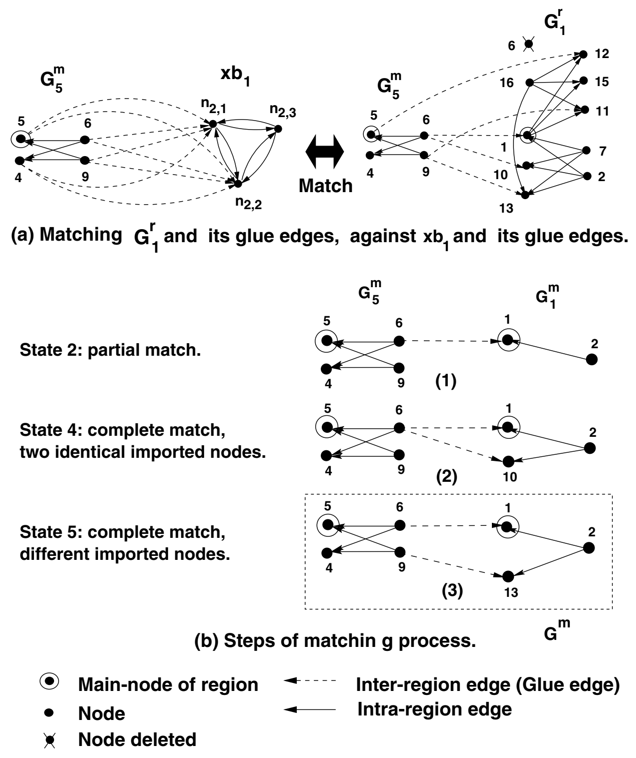
Step 2:
The AQL query aql is generated using the techniques discussed in section 5. The query consists of l abstract blocks, each corresponding to one or more regions of $S(G)$, along with the size ranges for abstract blocks and abstract links. An example was discussed in *definition 4*. The AQL query is expanded into *partial pattern graph* G^{p_i} at each phase i of the matching process.

Step 3:
The whole matching process is divided into l phases, where l is the number of abstract blocks (i.e., Modules or Subsystems) in aql. At each phase i ($i \leq l$), the search algorithm generates a new stage S_i (search tree) of the multi-stage search space and tries to find a path from the root of the tree to one of its leaves that produces a partial matching with locally minimum graph distance between two subgraphs G^{p_i} and G^{m_i}, so that $dist(G^{p_i}, G^{m_i}) \leq d_t$. Where, G^{p_i} and G^{m_i} are *partial pattern graph* and *partial matched graph* at phase i, respectively, and d_t is the *top cost.*

The generation of the whole search space in each stage is a combinatorial problem with exponential complexity (with respect to the number of the graph nodes). Therefore, for large graphs only a small portion of the search space must be generated which leads us to an optimal solution. We use the *branch and bound* search algorithm which expands the tree from a node with the minimum matching cost. In order to further limit the size of the search space, a *forward checking* mechanism is used that prunes the search tree based on early checking of the minimum range constraint (related to the cost of inter-region edge insertion c^{out}_{ei} discussed in section 6.2).

In order to explain the pattern matching process, we consider the input graph G in Figure 2 which has been decomposed into region database $S(G)$ including the regions G^r_5 and G^r_1. Based on the guidelines discussed in section 5, the user defines the AQL query aql for a part of the graph G, where the regions G^r_5 and G^r_1 have been assigned to the abstract blocks ab_5 and ab_1, respectively (Figure 4).

The pattern matching algorithm has two phases. In the first phase, the algorithm generates the partial pattern graph G^{p_1} (i.e., expanded block xb_5) and matches it against the region G^r_5 of the source graph. The result of the matching process, G^m_5, is illustrated in Figure 4.



G_5^m
xb_1
G_1^r
$n_{2,1}$
$n_{2,3}$
$n_{2,2}$
Match
(a) Matching G_1^r and its glue edges, against xb_1 and its glue edges.
State 2: partial match.
State 4: complete match, two identical imported nodes.
State 5: complete match, different imported nodes.
(1)
(2)
(3)
(b) Steps of matchin g process.
G^m
Main-node of region
Node
Node deleted
Inter-region edge (Glue edge)
Intra-region edge

Figure 5. Matching a subgraph of input graph G (Fig. 2) against pattern graph G^p (Fig. 4).

In the second phase, Figure 5(a), the algorithm incrementally matches "region G_1^r and its glue edges to G_5^m" against "expanded block xb_1 and its glue edges to G_5^m". The graph at the right part of Figure 5(a) illustrates the interaction between the current region G_1^r and the matched sub-region G_5^m based on the existing edges in the source model graph G. In this phase, the user also decide to assign node 2 from region G_1^r as a fixed node to appear in the result of the matching. The steps of the matching process and their correspondences with the *state space* are shown in Figure 5(b).

The search algorithm generates the second stage of the multi-stage state space (search space), shown in Figure 6. In this search space, each node (state) contains a fixed part (shown as black rectangles) related to the matched sub-region G_5^m and a changing part related to matching the nodes between G_1^r and xb_1.

The cost of graph edit operations up to a particular state (node) plus an *underestimate cost* of edit operations for the future matches along that path to a leaf node is shown inside each node, and the node number is shown outside the node. Since the cost of edit operations increases by matching more nodes along a path, an underestimate cost for the

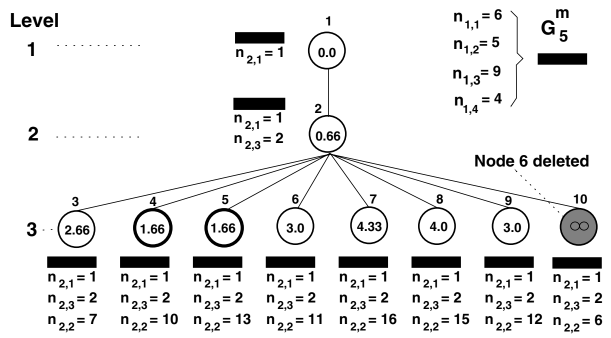


Figure 6. Second stage of multi-stage state space.

future node matching is used to allow the branch and bound algorithm proceed towards a solution and expand the paths with optimal cost. The underestimate cost, denoted as c_u, is the minimum cost of matching a node and is equal to the cost of an intra-region edge insertion c_{ei}^{in} with maximum association value in the corresponding region. In this example, the underestimate cost $c_u = \frac{1}{3}$, where 3 is the maximum association degree in the current region, i.e., G_1^r.

The node matching in Figure 6 is shown as $n_{x,y} = k$, where $n_{x,y}$ and k are the nodes of xb_1 and G_1^r, respectively. At level 2 of the tree, the algorithm uses "*edge-sink-change*" to match the seed-node 2 of G_1^r against a proper node of the expanded block xb_1 that lacks any inter-region edge (i.e., $n_{2,3}$). Therefore, the search tree is pruned to exclude the matching of the remaining nodes of G_1^r.

At the end of the matching process, the branch and bound algorithm finds the leaf nodes 4 and 5 with minimum graph edit costs which generate the matched graphs shown in parts 2 and 3 of Figures 5(b) with $dist(G^p, G^m) = 1.66$. The costs inside the nodes have been calculated based on the cost of graph edit operations discussed in section 6.2. However, the matching at state 4 is not a valid solution, since node 6 has been considered twice as an imported node to the component G_1^m. Such situations are checked by the search algorithm in selecting the proper non-leaf node to expand the tree. Finally, the complete match at part 3 of Figures 5(b) is a valid solution of this matching process example.

The graph edit costs corresponding to some of the states in Figure 6 are shown below:

$$C_1 = 0.0$$
$$C_2 = C_1 + c_u + c_{ei}^{in} = 0.0 + \tfrac{1}{3} + \tfrac{1}{3} = 0.66$$
$$C_3 = C_2 + 4 * c_{ei}^{out} + 3 * c_{ei}^{in} = 0.66 + 1 + 3 * \tfrac{1}{3} = 2.66$$
$$C_4 = C_5 = C_2 + 3 * c_{ei}^{out} + 3 * c_{ei}^{in} = 0.66 + 0 + 3 * \tfrac{1}{3} = 1.66$$
$$C_6 = C_2 + 3 * c_{ei}^{out} + 3 * c_{ei}^{in} = 0.66 + 0 + (\tfrac{1}{3} + 2 * 1) = 3.00$$
$$C_7 = C_2 + 4 * c_{ei}^{out} + 4 * c_{ei}^{in} = 0.66 + 1 + (2 * \tfrac{1}{3} + 2 * 1) = 4.33$$
$$C_{10} = C_2 + c_{nd} = 0.66 + \infty = \infty$$

In C_{10}, the cost of deleting node 6 from G_1^r is top cost, as defined for c_{nd} in section 6.2, since node 6 is a *matched-linked* node in G_5^m.

The inexact matching algorithm also handles the issues regarding to: i) resolving *shared nodes* in the matched sub-regions by means of a consistency checking algorithm after each phase, and ii) *backtracking* from one stage of the multi-stage search space to the previous stage, if the current stage can not produce a matching solution[8].

8 Experiments

In this section, the experimental results of the proposed architectural recovery technique are presented by analyzing the graphics editor *Xfig.3.2.3* using the developed prototype tool Alborz [13]. Our experimentation platform consists of a Sun Ultra 10 (440MHZ, 256M memory, 512M swap disk). In this framework, the user and tool cooperate in performing hierarchical and incremental recovery/restructuring at the system and subsystem levels as follows:

- *System level analysis*: at this level the entity relationships are of the form *"file use resource"*, where *"use"* indicates any kinds of references that a function inside the *file* performs to the *resource*, and a *resource* is a function, an aggregate type, or a global variable. In this analysis, the whole system of files is decomposed into a number of subsystems of files that interact via import/export of resources.

- *Subsystem level analysis*: at this level the entity relationships are of the forms *"func call-F func"*, *"func use-T type"*, and *"func use-V var"*. In this analysis, a subsystem from the system level analysis above is decomposed into a number of modules of functions, types and variables, where the modules interact via import/export of functions, types, and variables.

In a typical scenario, the user first generates an AQL query for a partial architectural pattern of the system and performs pattern matching. Then iterates this process by increasing (and enhancing) the AQL query and performing pattern matching to cover a major part of the system. At this point, the user may shift to the *distribution phase* (section 3) and perform some repair operations such as: i) distributing some of the unresolved entities among their closest blocks; ii) generating user-defined *concrete blocks* in order to accommodate the rest of the un-resolved entities into blocks; and iii) relocating entities among the blocks to adjust the block and link sizes.

[8]Because of space limitation, the mechanisms for *sharing-node resolution* and *backtracking* are not discussed here.

8.1 System analysis of Xfig

The Xfig system is an interactive drawing tool which runs under X Windows [1]. Xfig consists of 75 KLOC of source code written in C, distributed into 100 source files, 75 include files, 1662 functions, 1356 global variables, and 37 aggregate types. It takes 15 minutes to parse Xfig using a parser written in Refine C, and to construct an annotated AST in the Refine's database. The Apriori algorithm requires approximately 10 minutes to build the region spaces for system-level analysis.

The Xfig system lacks any documentation on the structure or implementation, however, a consistent naming convention is used throughout the system files [19]. This naming convention includes: d_* files relate to drawing shapes; e_* files relate to editing shapes; f_* files have file-related procedures; u_* files are utilities for drawing or editing shapes; and w_* files have X11 window calls in them to do all of the window-related functions.

Pattern generation
Figure 7(a) demonstrates the initial architectural pattern graph for a part of the Xfig system consisting of four subsystems. The subsystems S1 to S4 represent the *editing, utility, windowing*, and *file* subsystems of Xfig, respectively. The links represent the number of actual imported/exported functions between subsystems. The files in Figure 7(a) have been obtained by analyzing the component association graph [14] of Xfig to find the most important file(s) in each subsystem, as the core of the subsystem. Other alternatives have been discussed in section 5.

The graph representation of the AQL query, derived from the initial pattern graph, is shown in Figure 7(b). The core files (as main-seeds) in each subsystem of part (a) are used to assign the graph regions for each abstract block of the AQL query. In the query, two abstract links ?R1 and ?R3 connect S1 and S2 in order to provide more interaction between the *editing* subsystem and *utility* subsystem as suggested by the above description for Xfig files. We also restrict the interaction between the *editing* subsystem and *windowing* subsystem using the abstract link ?R4.

Knowledge incorporation
As mentioned above, the Xfig system files have been logically classified into subsystems based on naming conventions. Among other subsystems, two logical subsystems "editing shapes" and "utility" are highly related. A typical clustering algorithm may gather the files in each subsystem that only satisfy the cohesion property for each subsystem. However, the result has no correspondence with the maintainer's knowledge about the level of interaction between these systems.

Using the pattern-based architectural recovery technique

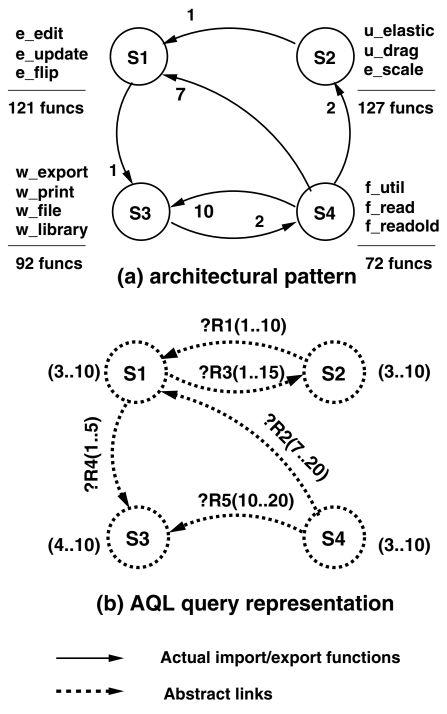


Figure 7. The initial architectural pattern of a part of Xfig system (section 5).

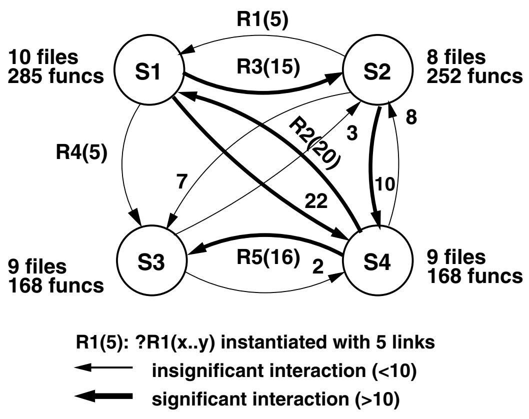


Figure 8. The concrete architecture (matched graph) of Xfig, resulted from matching the abstract graph (AQL query) with the source model graph.

proposed in this paper, the maintainer can incorporate the knowledge about the system domain and document into the process of re-modularization. In the case of Xfig system, the maintainer can define an abstract group of links between these two subsystems to be instantiated with a large number of single links. The pattern matching engine then tries to gather the files in these two subsystems, so that each subsystem is both cohesive and highly related to other subsystem, hence satisfying the requirements from the system's document.

Pattern matching

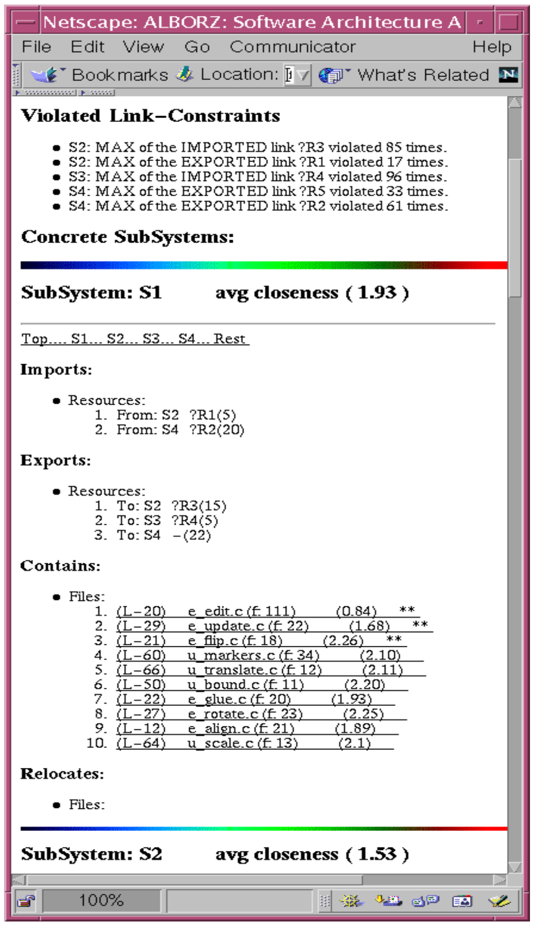
Figure 8 illustrates the result of matching the AQL query with the Xfig source model graph. For this query, all the node and link constraints that we defined in the AQL query have been satisfied and four concrete subsystems have been generated. The interaction between S1 and S2 has been increased and the interaction between S1 and S3 has been restricted. The links in Figure 8 that are not shown in Figure 7(b), correspond to *?IR* and *?ER* in the AQL query (section 4). These links are used to enhance the abstract graph of the system to run another matching. For example, in the next AQL query we may want to add abstract links (S1, S4) and (S2, S4) to control the interactions between the corresponding subsystems. This is because in the concrete architecture of Figure 8 significant interactions are shown between S4 and each of S1 and S2 which have not been specified as part of the pattern in the AQL query.

Figure 9 illustrates the detailed result of the pattern matching for subsystem S1 that abstractly shown in Figure 8. The top part of Figure 9 indicates the violated abstract link constraints which can be used to locate the critical links to be instantiated, in order to enhance the AQL query.

The contents and links of individual subsystems can be viewed by clicking on the menu list: *Top...S1...S2...S3...S4...Rest.* The subsystem S1 contains 10 files with main-seeds that are labeled with "**". Each file is accompanied with the number of its functions (e.g., file *e_edit.c* has 111 functions), and a closeness value to other files in that subsystem (e.g., 0.84). The closeness value is computed as the average association values and is used for entity distribution and relocation. The group links in import/export parts can also be viewed as individual links with detailed information about each function with a hypertext link to the source code.

In subsystem S1, 60% of the files are related to *editing* Xfig shapes and 40% of the files are related to utility files for editing Xfig shapes, hence, producing a cohesive subsystem.

Other experiments that relate to the Precision and Recall characteristics of the matching engine indicate that the matching engine can achieve a 50% Recall for a 85% Precision, and a 75% Recall for a 68% Precision. The above Precision and Recall have been measured by analyzing the CLIPS system and comparing the obtained results by the query against the architectural manuals of the system.



```
Violated Link–Constraints

  • S2: MAX of the IMPORTED link ?R3 violated 85 times.
  • S2: MAX of the EXPORTED link ?R1 violated 17 times.
  • S3: MAX of the IMPORTED link ?R4 violated 96 times.
  • S4: MAX of the EXPORTED link ?R5 violated 33 times.
  • S4: MAX of the EXPORTED link ?R2 violated 61 times.

Concrete SubSystems:

SubSystem: S1        avg closeness ( 1.93 )

Top.... S1... S2... S3... S4... Rest

Imports:

  • Resources:
       1. From: S2  ?R1(5)
       2. From: S4  ?R2(20)

Exports:

  • Resources:
       1. To: S2  ?R3(15)
       2. To: S3  ?R4(5)
       3. To: S4  –(22)

Contains:

  • Files:
       1. (L–20)   e_edit.c (f: 111)        (0.84)    **
       2. (L–29)   e_update.c (f: 22)        (1.68)    **
       3. (L–21)   e_flip.c (f: 18)        (2.26)    **
       4. (L–60)   u_markers.c (f: 34)        (2.10)
       5. (L–66)   u_translate.c (f: 12)        (2.11)
       6. (L–50)   u_bound.c (f: 11)        (2.20)
       7. (L–22)   e_glue.c (f: 20)        (1.93)
       8. (L–27)   e_rotate.c (f: 23)        (2.25)
       9. (L–12)   e_align.c (f: 21)        (1.89)
      10. (L–64)   u_scale.c (f: 13)        (2.1)

Relocates:

  • Files:

SubSystem: S2        avg closeness ( 1.53 )
```

Figure 9. The result of architectural recovery of the Xfig system using AQL query.

9 Conclusion

In this paper we presented a framework for software architecture recovery based on an inexact graph matching technique. The high-level design of a system is defined using a graph pattern language denoted as architectural query language. Based on the association property among the system entities, the software system is also presented as a collection of graph regions which are then stored in a database. The inexact graph matching engine uses a multi-phased branch and bound search with backtracking and forward checking. This search engine uses the graph distances derived from graph edit operations and the association properties in order to extract a subgraph of the system graph which matches with the user-defined pattern. We implemented a tool for the architectural analysis at system-level and module-level with features to meet the needs of the typical architectural recovery tasks. The experiments indicate that the proposed technique provides useful results to the maintainer and that it is scalable. On going work relates to the formal analysis of the matching process with respect to its time and space complexity. This work has been performed within the framework of a project funded by of the Consortium for Software Engineering Research, the Institute for Robotics and Intelligent Systems, and in cooperation with IBM Toronto Laboratory, Center for Advanced Studies.

References

[1] Xfig User Manual, Web site, URL = http://www.xfig.org/userman/.

[2] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, Chile, 1994.

[3] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[4] R. Chanchlani. Software architecture recovery and design using partitioning. Master's thesis, University of Waterloo, 1998.

[5] M. A. Eshera and K.-S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems Man and Cybernetics*, SMC-14(3):398–408, May/June 1984.

[6] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, et al. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[7] R. Kazman and M. Burth. Assessing architectural complexity. In *CSMR*, pages 104–112, 1998.

[8] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of IWPC'98*, pages 45–53, Ischia, Italy, 1998.

[9] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–503, May 1998.

[10] R. J. Miller and A. Gujarathi. Mining for program structure. *International Journal on Software Engineering and Knowledge Engineering*, 9(5):499–517, 1999.

[11] H. A. Muller, M. Orgun, et al. A reverse-engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[12] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion model: Bridging the gap between source and higher-level models. In *In proceedings of the 3rd ACM SIGSOFT SFSE*, pages 18–28, October 1995.

[13] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 115–116, Toronto, Canada, May 2001.

[14] K. Sartipi. A software evaluation model using component association views. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 259–268, Toronto, Canada, May 2001.

[15] K. Sartipi and K. Kontogiannis. Component clustering based on maximal association. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE'01*, Stuttgart, Germany, October 2001. (to appear).

[16] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architectural design recovery using data mining techniques. In *Proceedings of IEEE CSMR 2000*, pages 129–139, Zurich, Switzerland, Feb 29 - March 3 2000.

[17] K. Sartipi, K. Kontogiannis, and F. Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of IEEE IWPC 2000*, pages 37–47, Limerick, Ireland, June 10-11 2000.

[18] L. G. Shapiro and R. M.Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Matching Intelligence*, PAMI-3(5):504–519, September 1981.

[19] B. V. Smith. Xfig architecture, September 2000. Personal e-mail correspondence with author.

[20] S. G. Woods, A. Quilici, and Q. Yang. *Constraint-Based Design recovery for Software Reengineering: Theory and Experiments*. Kluwer Academic Publishers, 1998.