# On Modeling Software Architecture Recovery as Graph Matching [*]

Kamran Sartipi[1] Kostas Kontogiannis[2]

University of Waterloo
School 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 graph matching model for the software architecture recovery problem. Because of their expressiveness, the graphs have been widely used for representing both the software system and its high-level view, known as the conceptual architecture. Modeling the recovery process as graph matching is an attempt to identify a sub-optimal transformation from a pattern graph, representing the high-level view of the system, onto a subgraph of the software system graph. A successful match yields a restructured system that conforms with the given graph pattern. A failed match indicates the points where the system violates specific constraints. The pattern graph generation and the incrementality of the recovery process are the important issues to be addressed. The approach is evaluated through case studies using a prototype toolkit that implements the proposed interactive recovery environment.*

## 1 Introduction

Most approaches to software architecture recovery view the recovery process as: i) a pattern matching problem that models the recovery by identifying groups of system entities whose properties closely match with the user-defined queries [10, 6]; ii) a clustering problem that models the recovery by grouping the related parts of a software system into cohesive components [8, 9]; iii) a constraint satisfaction problem that models the recovery by identifying groups of entities that meet the conditions defined in a repository of plans [16]; iv) a lattice partitioning problem that models the recovery by classifying maximally related groups of entities that are arranged in a lattice [15]; or v) a composition and visualization problem that models the recovery by aggregating system entities into containment-hierarchy of components [4].

The reverse engineering community has also paid particular attention to the pattern matching approaches since they allow the use of domain knowledge and system documentation in composing the pattern, hence provide a user/tool cooperative environment for architectural recovery. Moreover, the software systems are intuitively represented as graphs and the reverse engineering community is on the verge of adopting a graph standard for information exchange among the existing reverse engineering tools [5]. This paper presents an approach to software architecture recovery that considers the high-level design of a system as a pattern graph, and models the recovery process as a graph pattern matching problem that matches such a high-level pattern graph of the system with an entity-relationship graph representation of the source-code system entities.

The motivation for this research stems from the lack of a reflective and uniform model for pattern-based software architectural recovery, whereby the software system, architectural pattern, and pattern matching process, are all uniformly represented using a graph formalism, and the recovered architecture conforms with detailed constraints of the architectural pattern.

The remaining sections of this paper are organized as follows. The related work is discussed in Section 2. Section 3 presents the proposed environment for architectural recovery. Sections 4 and 5 represent the software system and its architectural pattern. Sections 6 and 7 discuss the graph matching process and modeling. Section 8 presents the tractability of the matching process. Section 9 provides the steps for pattern generation, and Section 10 presents the architectural recovery case studies.

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

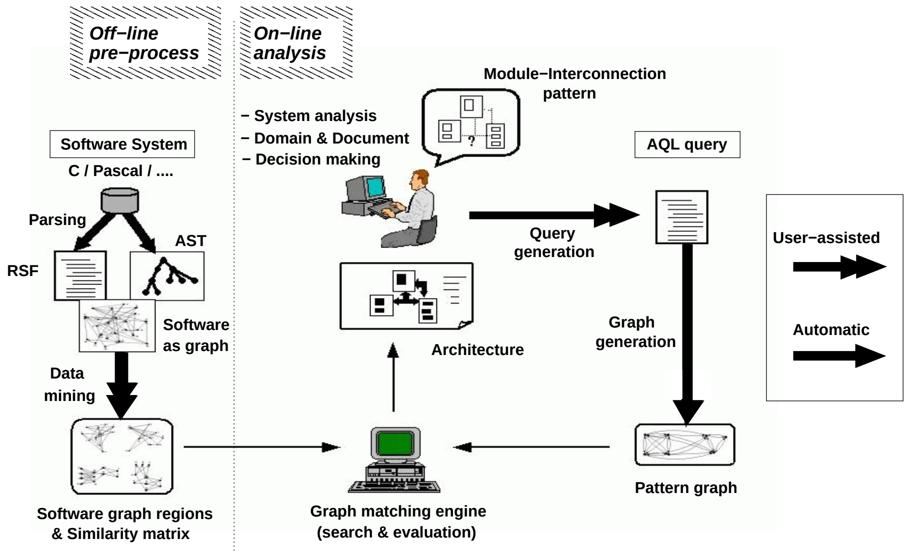


**Figure 1. The interactive environment for the proposed pattern-based software architecture recovery.**

## 2 Related work

The related approaches to this work include the followings. In *Dali* [7] a pattern consists of a collection of SQL queries that collect the architectural components and their derived relations, whereas the approach in this paper presents a modular pattern of the software system using the AQL queries. In [6], the user defines an architectural pattern or style as a graph of components and connectors representing single elements and uses approximate matching to identify the pattern in the software system. In contrast our approach defines a macroscopic pattern on the groups of system entities and the interaction among the groups of entities. In [10], a software reflexion model is proposed to assist the user in testing his/her mental model of the system using textual declarative forms. In contrast, our approach uses a structured query as pattern and architectural constraints to be satisfied in the recovered architecture, as opposed to checking the validation of the facts in the pattern. The approach in this paper also relates to our previous work on a graph pattern matching approach to software architecture recovery [14]. Specifically, the contributions of this paper include: i) providing representation models for the software system being analyzed and the high-level pattern of components and constraints; ii) modeling the graph matching process by a set of recursive graph equations; and iii) steps for generating the architectural pattern graph.

## 3 Proposed environment for recovery process

Despite several attempts for automating the architectural recovery process (i.e., clustering) it is generally accepted that a fully-automated technique is not feasible. It is rather difficult to extract the architecture of a large system at once, hence, the architectural recovery should be an incremental process. Software systems usually consist of patterns in their design which form the basis for the recovery process. Most recovery processes focus on the structural properties of a system, ignoring the high-level behavior of the system. Finally, the role of the user is increasingly important in incorporating the domain knowledge and system documents into the recovery process. Based on the above discussion, this paper defines the software architectural recovery problem as:

> *devising a tractable methodology and the supporting tools for interactively and incrementally extracting a system's structure using domain and system knowledge*.

We propose an interactive reverse engineering environment for incremental recovery and evaluation of the architecture of a software system in the form of cohesive modules (or subsystems) that comply with the constraints of a user-defined pattern.

Figure 1 illustrates the different parts of the proposed interactive architectural recovery environment where the thick arrows signify the automatic or user-assisted processes in the environment; boxes represent the different forms of information in the environment; the thin arrows indicate the inputs and output of the graph matching engine; and the user is the high-level decision maker that produces a mental model of the architecture and verifies the result of recovery. The environment consists of an *off-line pre-process* phase and an *on-line* analysis phase.

During the off-line information extraction phase, the software system, written in a procedural language such as C, is parsed and presented as an attributed relational graph whose nodes and edges conform with a domain model that is suitable for architectural recovery. Such a domain model provides programming language independence for the recovery process. The graph representation of the software system is further processed and is divided into a collection of subgraphs, where the appropriate subgraphs are selected by the graph matching process as the subspaces for recovery of the system components. Also, a similarity matrix is generated that contains the association-based similarity values between every two system entities to be used for recovery of cohesive components.

During the on-line analysis phase, the user defines a graph-based architectural pattern of the system modules (subsystems) and their interactions based on: domain knowledge, system documents, or tool-provided system analysis information. In an iterative recovery process, the user constraints the architectural pattern and the tool provides a decomposition of the system entities into modules or subsystems that satisfy the constraints. In this approach, the architectural pattern is viewed as a graph of modules (or subsystems) and interconnections, where each module (one node of graph) represents a group of placeholders for the system entities (i.e., functions, types, variables) to be instantiated, and each bundle of interconnections (one edge of graph) between two modules represents data-/control-dependencies between two groups of placeholders in two modules. The minimum/maximum sizes and the types of both placeholders and the interconnections are considered as free parameters to be decided by the user (respecting the allowed relation between two entities).

This yet un-instantiated module-interconnection representation (can be referred to as *conceptual architecture*) is directly defined for the tool, using a proprietary language that we call *Architecture Query Language* (AQL) that is discussed in [14].

**Pattern-graph**: the AQL query is incrementally expanded to generate a pattern-graph that represents a macroscopic view and structural constraints for a part or the whole of the system architecture to be recovered. The task of the tool is then to search through the software system (again represented as a graph of system entities and relationships) to find an sub-optimal match between the pattern-graph and the graph of the system. The architectural recovery is performed at two levels of abstraction. At the *file-level*, the software system is decomposed into a number of subsystems of files, and at the *function-level* each recovered subsystem can be decomposed into a number of modules of functions, datatypes, and variables.

## 4 Software system representation

In this approach the software system and the architectural pattern are presented using the *attributed relational graph* notion defined in [3].

**Source-graph**: the software system is represented by the source-graph $G^s = (N^s, R^s)$, where the nodes ($n_j$) represent files, functions, datatypes, and variables and the edges ($r_y$) represent *contain* and *use* relationships. The nodes and edges comply with the specific domain model, namely *abstract domain model*, illustrated in Figure 2(b). In this model different types of entities, i.e., *File-abs, Function-abs, Type-abs, Variable-abs*, are a subset of the types of entities in the software system's source-code. Where, *Function-abs* denotes functions, *Type-abs* denotes aggregate/array types, and *Variable-abs* denotes global variables in the software system. Also, functions, datatypes, and variables are called *simple entities*, and files are called *composite entities* such that a file may contain zero or more simple entities.

Each relation in the abstract domain model, i.e., *use-F, use-T, use-V, cont-R, use-R, imp-R, exp-R*, is an aggregation of one or more relations in the software system's source-code. The relations *use-F, use-T, use-V* are defined such that the implementation of a function-abstraction (i.e., a function in the source-code) *calls* another function; updates/reads a variable whose type is an aggregate/array type; or updates/reads a global variable, respectively. The relation *cont-R* (i.e., contained resource) is defined such that, each simple entity can be contained in only one file; the simple entities in the library files are contained in the file abstractions based on the maximum frequency of using the simple entities by the files. In this context, the externally defined library files and their contained simple entities are not considered. The relation *use-R* (i.e., use resource) is defined such that a file contains a function and that function uses a simple entity as described above. The relation *imp-R* (i.e., import resource) is defined such that a simple entity is contained in another file but is used by the subject file (and vice versa for *exp-R*). Based on the above abstract domain model the software system is parsed and represented as an attribute relational graph whose abstraction-level is suitable for architectural analysis, as it is illustrated in Figure 2(c). Two labeling functions $\mu^s$ and $\epsilon^s$ in Figure 2(c) are used to return the important attribute values of the nodes and edges of the source-graph in the form of a list of "attribute, attribute-value" pairs.

In Figure 2(b), the class *Entity-abs* (*Relation-abs*) presents the common attributes that are inherited by every entity (relation) in the abstract domain model. These attributes identify a source-code construct (e.g., *definition, declaration, statement, function-call, assignment*) that implement a specific entity or relation. Each relation in the abstract domain model is an object of an "association class" and contains the attributes "from" and "to" denoting the source and destination entity for that relation.

**Reducing the search space:** the source-graph $G^s$ pro-

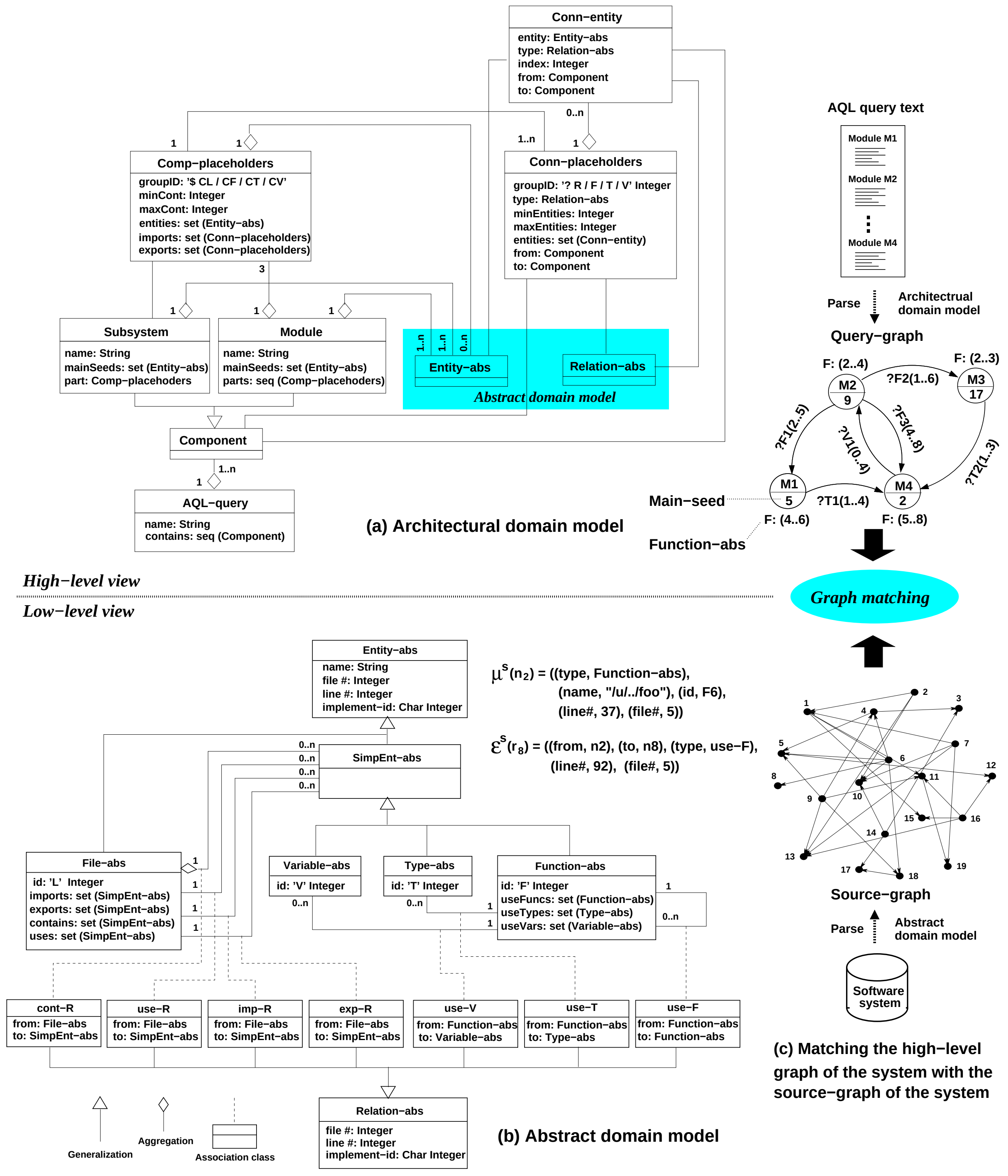

Conn−entity
entity: Entity−abs
type: Relation−abs
index: Integer
from: Component
to: Component
Comp−placeholders
groupID: '$ CL / CF / CT / CV'
minCont: Integer
maxCont: Integer
entities: set (Entity−abs)
imports: set (Conn−placeholders)
exports: set (Conn−placeholders)
Conn−placeholders
groupID: '? R / F / T / V' Integer
type: Relation−abs
minEntities: Integer
maxEntities: Integer
entities: set (Conn−entity)
from: Component
to: Component
Subsystem
name: String
mainSeeds: set (Entity−abs)
part: Comp−placeholders
Module
name: String
mainSeeds: set (Entity−abs)
parts: seq (Comp−placeholders)
Entity−abs
Relation−abs
Abstract domain model
Component
AQL−query
name: String
contains: seq (Component)
(a) Architectural domain model
AQL query text
Module M1
Module M2
Module M4
Parse
Architectrual domain model
Query−graph
F: (2..4)
F: (2..3)
?F2(1..6)
?F1(2..5)
?F3(4..8)
?V1(0..4)
?T2(1..3)
?T1(1..4)
Main−seed
Function−abs
F: (4..6)
F: (5..8)
Graph matching
High−level view
Low−level view
Entity−abs
name: String
file #: Integer
line #: Integer
implement−id: Char Integer
SimpEnt−abs
File−abs
id: 'L' Integer
imports: set (SimpEnt−abs)
exports: set (SimpEnt−abs)
contains: set (SimpEnt−abs)
uses: set (SimpEnt−abs)
Variable−abs
id: 'V' Integer
Type−abs
id: 'T' Integer
Function−abs
id: 'F' Integer
useFuncs: set (Function−abs)
useTypes: set (Type−abs)
useVars: set (Variable−abs)
$\mu^s(n_2)$ = ((type, Function−abs), (name, "/u/../foo"), (id, F6), (line#, 37), (file#, 5))
$\varepsilon^s(r_8)$ = ((from, n2), (to, n8), (type, use−F), (line#, 92), (file#, 5))
cont−R from: File−abs to: SimpEnt−abs
use−R from: File−abs to: SimpEnt−abs
imp−R from: File−abs to: SimpEnt−abs
exp−R from: File−abs to: SimpEnt−abs
use−V from: Function−abs to: Variable−abs
use−T from: Function−abs to: Type−abs
use−F from: Function−abs to: Function−abs
Relation−abs
file #: Integer
line #: Integer
implement−id: Char Integer
Generalization
Aggregation
Association class
(b) Abstract domain model
Source−graph
Parse
Abstract domain model
Software system
(c) Matching the high−level graph of the system with the source−graph of the system

**Figure 2. Representing the architectural pattern and the software system as two graphs using the architectural domain model and abstract domain model, respectively.**

vides a search-space for the matching process. However, since even in a medium-size software system the number of entities and relationships that are generated are prohibitively high, any matching algorithm will be intractable.

**Matching phase**: to address the tractability of the matching process, the whole process is divided into $k$ incremental phases (as $k$ partial-matchings) where $k$ is the number of components to be recovered and the current matching phase is identified by "$i$" ($i \in$ [1.. No. of components]). Therefore, the recovery process performs a *multi-phase* matching.

**Source-region**: the search space for the matching process is divided into a collection of sub-spaces using data mining association relation defined in [14], where each sub-space is a subgraph of the source-graph $G^s$, namely a source-region $G^{sr}_j$. All the nodes in a source-region $G^{sr}_j$ are associated with a distinguished node $n_j$ in that region which is called the main-seed of the region. The group of source-regions $G^{sr}_j$'s in the source-graph $G^s$ are stored in a database and at the matching phase $i$ the user selects a source-region from the database to be matched with the incremental part of the pattern. Therefore, the source-region $G^{sr}_j$ is shown as $G^{sr}_{g(i)}$ where $g(i) = j$ is a function that maps the current matching phase (i.e., $i$) to the id-number of the selected source-region (i.e., $j$). At file-level analysis the source-region nodes are files functions, datatypes, variables, and at function-level analysis the source-region nodes are functions, datatypes, and variables.

## 5 Architectural pattern representation

The architectural pattern is represented by an *AQL query* that defines a macroscopic graph pattern for the software system to be matched against the source-graph $G^s$.

**Query-graph**: the syntactic constructs of an AQL query conform with the architectural domain model of Figure 2(a) that define a query-graph $G^q = (N^q, R^q)$. In this domain model, an AQL query, or equivalently a query-graph $G^q$, consists of a number of *abstract components*. Each abstract component (or simply a component) can be either a *subsystem* or a *module*. An abstract component is specified as a collection of placeholders. The interconnections among the abstract components are established by the means of *abstract connectors*, where an abstract connector is also specified by a number of placeholders. A **placeholder** is a node that can be matched with a system entity in the abstract domain model during the matching process. The user can constrain the minimum and maximum numbers of the matched placeholders and their types in the recovered components and connectors by formulating the AQL query.

Each abstract component contains one or more fixed entities, namely *main-seed*(s), that appear in the result of the recovery. The main-seed(s) for an abstract component determine the source-region(s) to be searched for recovery of that component.

In Figure 2(c), an AQL query with four modules is parsed and represented as a query-graph $G^q$ with four nodes that is interpreted as follows. The module M1 with main-seed node $n_5$ will be matched with minimum 4 and maximum 6 functions in the source-graph $G^s$ (presented as $F$:$(4..6)$), will import between 2 to 5 functions from module M2 (presented as $?F1(2..5)$), and will export between 1 to 4 datatypes to module M4 (presented as $?T1(1..4)$). The interaction of M1 with other modules is not restricted.

**Pattern graph generation:** the query-graph $G^q$ is used to derive a pattern-graph and an input-graph that are required for the incremental graph matching process. At matching phase $i$ the query-graph incrementally generates the pattern-graph $G^p_i$ as follows.

**Pattern-region**: the $i_{th}$ node of the query-graph $G^q$ (as the abstract-component $M_i$) is expanded into a pattern-region $G^{pr}_i$ through: i) generating the maximum number of placeholder nodes (or simply nodes) defined by $M_i$; and ii) connecting every node in the pattern-region to every other node in the pattern-region that are allowed based on the types of the nodes.

**Edge-bundle**: each edge of the query-graph $G^q$, e.g., edge with label $?Fx(min..max)$ and of type *use-F* is expanded into $max$ number of edge-bundles of the same type. Each edge-bundle connects every node from an already recovered component to one node (i.e, the common sink or source node) in the pattern-region $G^{pr}_i$ with respect to the direction of the query-graph edge. Initially, the first *max* nodes of the pattern-region are selected as the common sink/source nodes of the individual edge-bundles. However during the matching process the common sink/source node of an edge-bundle which is not matched yet can be redirected to another node of the pattern-region. Each edge-bundle corresponds to one node of the involving components to be imported or exported. The group of edge-bundles between the already recovered components (modules or subsystems) and the current pattern-region $G^{pr}_i$ is represented by $\mathcal{R}^{m \leftrightarrow pr_i}_i$[1].

The rationale for generating the edge-bundles is to allow every subset of the nodes in a source component to be connected to every subset of the nodes in the destination component, according to the constraints defined in the

[1]A group of *connector-edges* is denoted by $\mathcal{R}^{G_1 \leftrightarrow G_2}$ and represent a group of edges that connect two graphs $G_1$ and $G_2$. The connector-edges represent the interaction between two graphs in uni-directional (using $\leftarrow$ or $\rightarrow$) or bidirectional (using $\leftrightarrow$) mode. The connector-edges between a matched-graph $G^m_{i-1}$ (discussed later) and the source-region $G^{sr}_{g(i)}$ at phase $i$ are denoted by $\mathcal{R}^{m \leftrightarrow sr_i}_i$, and the edge-bundles between a matched-graph $G^m_{i-1}$ and the pattern-region $G^{pr}_i$ at phase $i$ are denoted by $\mathcal{R}^{m \leftrightarrow pr_i}_i$.

query-graph.

Figure 3(a) illustrates a query-graph $G^q$ with two nodes that represent two abstract-components M1 and M2 as modules, and an edge that represents an abstract-connector ?F1:(1..2). This query-graph has the required information to generate the incremental parts of both the pattern-graph $G_2^p$ and input-graph $G_2^I$ at phase 2 of the matching process. The node M2 with node constraint F:(2..3) generates a pattern-region $G_2^{pr}$ with maximum of three placeholder-nodes ($n_{2,1}$ to $n_{2,3}$) of types *Function-abs* and the edge ?F1:(1..2) generates two edge-bundles represented by $\mathcal{R}_2^{m \leftrightarrow pr_2}$ in Figure 3(b). Also, the node M2 identifies the selected source-region with main-seed node 1 in Figure 3(c), corresponding to $G_{g(2)}^{sr}$.

## 6 Graph matching process

A pattern-graph $G_i^p$ by its definition is composed of a number of smaller patterns (i.e., individual pattern-regions $G_i^{pr}$ at different matching phases $i$). This composition property allows to manage the complexity of the matching process of a large source-graph by applying it on a region-by-region basis.

**Matched-graph**: at matching phase $i$, the matching process computes a sub-optimal match, namely the matched-graph $G_i^m$ between a pattern-graph $G_i^p$ that originates from a query-graph $G^q$ and an input-graph $G_i^I$ that originates from the source-graph $G^s$. The obtained result must conform with the constraints of the query-graph with respect to the node and edge size-ranges. In Figures 3(b),(c),(d) an example of matching process at phase 2 is illustrated, where the "pattern-region $G_2^{pr}$ and its edge-bundles $\mathcal{R}_2^{m \leftrightarrow pr_2}$" from the pattern-graph $G_2^p$ are matched against the "source-region $G_{g(2)}^{sr}$ and its connector-edges $\mathcal{R}_2^{m \leftrightarrow sr_2}$" from the input-graph $G_2^I$. The result of matching is the "*matched-region* $G_2^{mr}$ and its connector-edges $\mathcal{R}_2^{m \leftrightarrow mr_2}$" from the matched-graph $G_2^m$. At the next iteration, the matched-graph $G_2^m$ is used to build the pattern-graph $G_3^p$ and input-graph $G_3^I$ to be matched at phase 3 (not applicable in this example).

## 7 Graph model of the matching process

The graph summation notations "plus $+$" for connecting two graphs, and "oplus $\oplus$" for connecting a graph to a group of connector-edges are used to model the whole incremental pattern matching process in terms of the recursive graph algebraic equations. Figure 4 illustrates the proposed graph matching model where $i$ is the matching phase and $|N^q|$ is the number of nodes in the query-graph $G^q$. In this context, the matched-graph at phase zero $G_0^m$ is defined as a $Nil$ graph with zero number of nodes and edges, and when $i = |N^q|$ then $G_i^I = G^I, G_i^p = G^p, G_i^m = G^m$, and the matching process terminates. The approximate matching process (equation 4) aims to compute a match between the pattern-graph $G_i^p$ and the input-graph $G_i^I$ by comparing different subgraphs $G_i^x$ of the input-graph $G_i^I$ against a transformed version of the pattern-graph $G_i^p$. A subgraph $G_i^x$ with minimum **graph distance** to the pattern-graph $G_i^p$ (a graph distance is computed as the total cost of a number of transformations on the pattern-graph $G_i^p$ such as node or edge deletion/insertion) is the solution of the matching process which is called the matched-graph $G_i^m$.

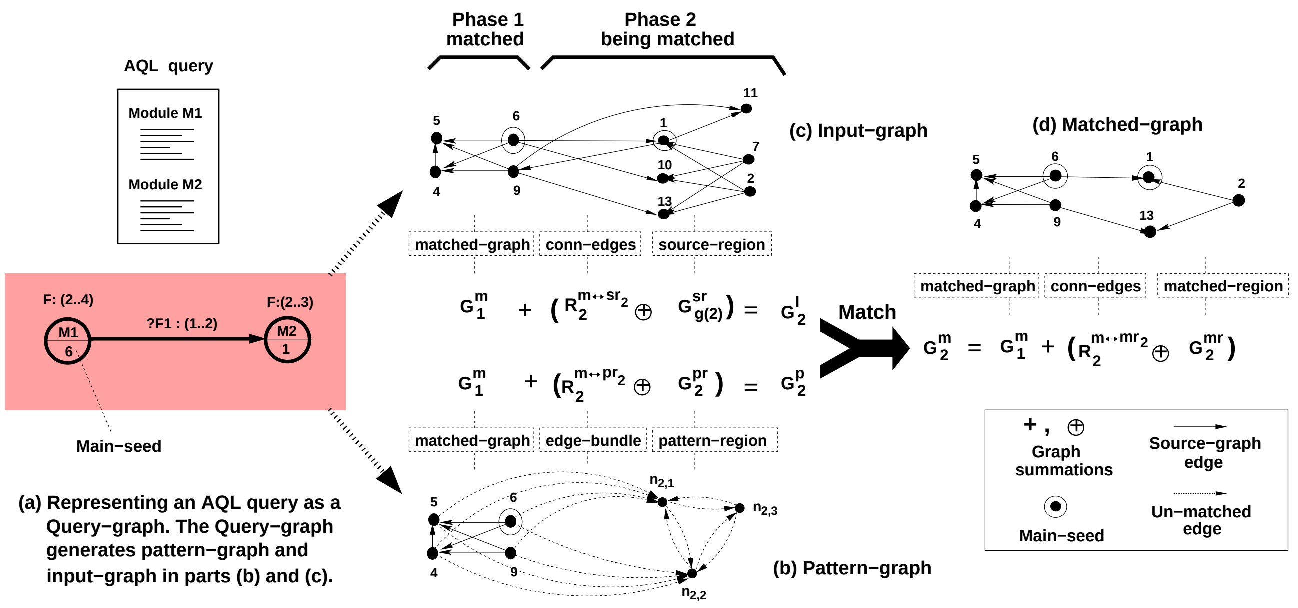
1. $G_0^m = \phi, \qquad i \in [1 \,..\, |N^q|]$
2. $G_i^I = G_{i-1}^m \;+\; (\mathcal{R}_i^{m \leftrightarrow sr_i} \;\oplus\; G_{g(i)}^{sr})$
3. $G_i^p = G_{i-1}^m \;+\; (\mathcal{R}_i^{m \leftrightarrow pr_i} \;\oplus\; G_i^{pr})$
4. $G_i^m = match(G_i^p, G_i^I) \mid$
   $dist(G_i^p, G_i^m) = Min\,\{dist(G_i^p, G_i^x) \mid G_i^x \subset G_i^I\}$
5. $dist(G_i^p, G_i^x) =$
   $\sum_{j=1}^{|N_i^{pr}|} (C_{in}^{ed} + C_{out}^{ed} + C_{out}^{ei})|_{n_{i,j}\ matches\ with\ n_{k_j}}$
6. $G_i^m = G_{i-1}^m \;+\; (\mathcal{R}_i^{m \leftrightarrow mr_i} \;\oplus\; G_i^{mr})$
7. $G_i^I = G^I, \quad G_i^p = G^p, \quad G_i^m = G^m \qquad if\ \ i = |N^q|$

**Figure 4. The recursive equations for the proposed multi-phase, incremental, and approximate graph matching process.**

### 7.1 Cost evaluation for graph matching

The equation 5 in Figure 4 defines the distance between the pattern-graph $G_i^p$ and a subgraph $G_i^x$ of the input-graph $G_i^I$, as the sum of the graph transformation costs for matching all the placeholder-nodes inside the pattern-region $G_i^{pr}$ with nodes from the subgraph $G_i^x$. In this distance, the costs for deleting the edges from inside the pattern-region ($C_{in}^{ed}$), the cost for deleting the edges from the edge-bundles ($C_{out}^{ed}$), and the cost for inserting the connector-edges ($C_{out}^{ei}$) are computed. We note that, only the incremental parts of the pattern-graph $G_i^p$ and the input-graph $G_i^I$ at matching phase $i$ are being matched and the matched-graph $G_{i-1}^m$ at phase $i-1$ is fixed, as shown in Figure 3(b) and (c). In the following these costs are defined.

**Inside-edge deletion cost,** $C_{in}^{ed}$: this cost is defined in Figure 5(a) with the objective of recovering cohesive components as the matched-regions $G_i^{mr}$'s at different



$G_1^m + (R_2^{m\leftrightarrow sr_2} \oplus G_{g(2)}^{sr}) = G_2^I$

$G_1^m + (R_2^{m\leftrightarrow pr_2} \oplus G_2^{pr}) = G_2^p$

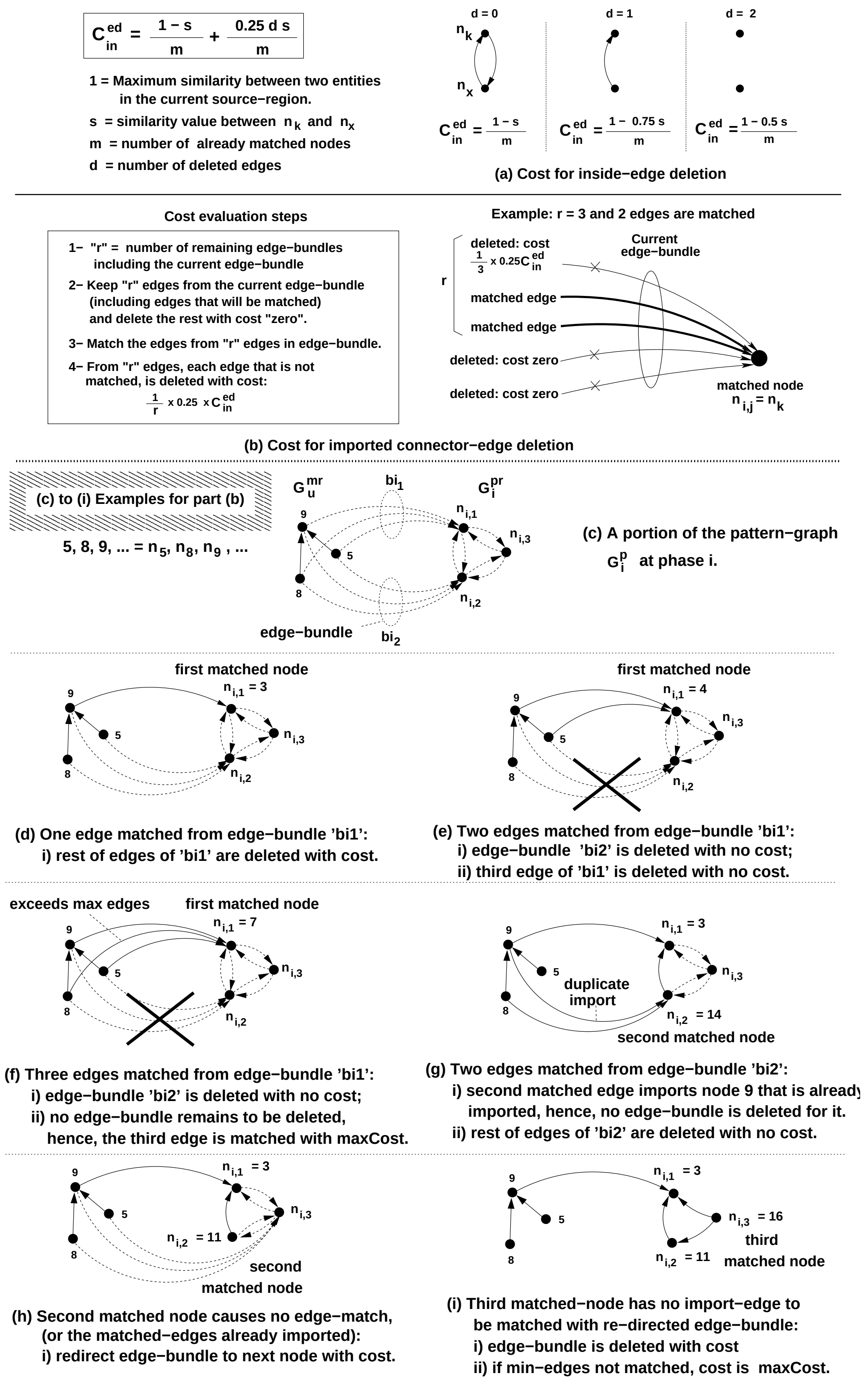
$G_2^m = G_1^m + (R_2^{m\leftrightarrow mr_2} \oplus G_2^{mr})$

**Figure 3. The graph pattern-matching process iteratively matches a pattern-graph with an input-graph and yields a matched-graph as the recovered architecture for the current matching phase.**

matching phases. This cost has two sub-costs. The first sub-cost $\frac{1-s}{m}$ is denoted as the *default dissimilarity* value between the candidate-node $n_k$ and each matched node in $G_i^{pr}$ where $s$ is the similarity value between two nodes according to a similarity metric such as the association similarity metric defined in [13]. The second sub-cost $\frac{0.25d\ s}{m}$ depends on the number of deleted edges "$d$" between two nodes. The coefficient "0.25" indicates the significance of each missing edge compared to the dissimilarity value between two nodes. Hence, each missing edge adds a value of $\frac{0.25s}{m}$ to the default cost value $\frac{1-s}{m}$, and at worst case (two edges deletion) the dissimilarity value doubles. Therefore, increasing the coefficient $0.25$ causes that the missing edge to become more important than the dissimilarity value, and vice versa. The costs for different cases of inside-edge deletion are shown in Figure 5(a).

**Connector-edge deletion cost,** $C_{out}^{ed}$: this cost is defined in Figure 5(b) based on the number of *remaining* edge-bundles "$r$" during the matching process at phase $i$ where the placeholder-node $n_{i,j}$ is to be matched with candidate-node $n_k$ from the selected source-region $G_{g(i)}^{sr}$. At this time, the placeholder-nodes $n_{i,1}, n_{i,2}, .., n_{i,j-1}$ in pattern-region $G_i^{pr}$ and their connector-edges (to previously matched-regions $G_u^{mr}$, $u < i$) have already been matched. Initially, for each node to be imported or exported between the involving components one edge-bundle has been generated. As an example, for each abstract-connector with minimum cardinality 1 and maximal cardinality 3, three edge-bundles are initially generated. Therefore, during the matching process, for each node to be imported/exported one edge-bundle must be deleted. The number of remaining edge-bundles indicates the number of nodes that can still be imported/exported to reach to the maximum number of allowed connector-edges. To perform cost evaluation, the connector-edges are further classified into two groups "*imported*" and "*exported*". Because of the space limitation, we only discuss the cost for "imported connector-edge deletion" in this paper.

**Imported connector-edge deletion:** for importing each node (which is equivalent to matching an imported connector-edge whose source node has not been imported earlier) a whole imported edge-bundle must be deleted with no-cost. However, within the edge-bundle that is connected to the current placeholder node $n_{i,j}$, for each edge deletion a cost is applied. The cost evaluation steps along with an example are illustrated in Figure 5(b). In this cost, "$r$" is the number of remaining edge-bundles including the current edge-bundle, and "$r$" is equal to the difference between maximum number of allowed edges to be matched and the number of currently matched edges. The value of this cost depends on the success of the candidate-node $n_k$ in augmenting the number of matched imported edges to reach to its maximum number. Therefore, matching more imported edges by the current node means less cost. This cost is calculated based on the eligibility of the node to produce a cohesive module, by taking into account the "inside-edge deletion cost $C_{in}^{ed}$". The coefficient *"0.25"* has been empirically determined to give more weight for collecting a group of related nodes as opposed to satisfying the max number of



$$C^{ed}_{in} = \frac{1-s}{m} + \frac{0.25\,d\,s}{m}$$

1 = Maximum similarity between two entities in the current source−region.

s = similarity value between $n_k$ and $n_x$

m = number of already matched nodes

d = number of deleted edges

(a) Cost for inside−edge deletion

**Cost evaluation steps**

1− "r" = number of remaining edge−bundles including the current edge−bundle

2− Keep "r" edges from the current edge−bundle (including edges that will be matched) and delete the rest with cost "zero".

3− Match the edges from "r" edges in edge−bundle.

4− From "r" edges, each edge that is not matched, is deleted with cost: $\frac{1}{r} \times 0.25 \times C^{ed}_{in}$

(b) Cost for imported connector−edge deletion

(c) to (i) Examples for part (b)

5, 8, 9, ... = $n_5$, $n_8$, $n_9$ , ...

(c) A portion of the pattern−graph $G^p_i$ at phase i.

(d) One edge matched from edge−bundle 'bi1':
i) rest of edges of 'bi1' are deleted with cost.

(e) Two edges matched from edge−bundle 'bi1':
i) edge−bundle 'bi2' is deleted with no cost;
ii) third edge of 'bi1' is deleted with no cost.

(f) Three edges matched from edge−bundle 'bi1':
i) edge−bundle 'bi2' is deleted with no cost;
ii) no edge−bundle remains to be deleted, hence, the third edge is matched with maxCost.

(g) Two edges matched from edge−bundle 'bi2':
i) second matched edge imports node 9 that is already imported, hence, no edge−bundle is deleted for it.
ii) rest of edges of 'bi2' are deleted with no cost.

(h) Second matched node causes no edge−match, (or the matched−edges already imported):
i) redirect edge−bundle to next node with cost.

(i) Third matched−node has no import−edge to be matched with re−directed edge−bundle:
i) edge−bundle is deleted with cost
ii) if min−edges not matched, cost is maxCost.

**Figure 5. The cost evaluation for deleting edges that are located: i) inside the pattern-region, and ii) within the edge-bundles between a recovered module and the pattern-region.**

imported connector-edges.

However, this cost evaluation is applied only if the minimum (maximum) number of imported connector-edges for pattern-region $G_i^{pr}$ still can be reached (not exceeded) by the number of matched connector-edges for the current node-matching $n_{i,j}$ with $n_k$. Otherwise, the cost is *maxCost* and this node-matching will be discarded.

**Connector-edge insertion cost, $C_{out}^{ei}$:** If there exist edge-bundles between the pattern-region $G_i^{pr}$ and the matched-region $G_u^{mr}$ $(u < i)$ in the pattern-graph $G_i^p$ then the cost of inserting a connector-edge (in the same direction as the edge-bundles) in $\mathcal{R}_i^{mr_u \leftrightarrow pr_i}$ is *maxCost*. Otherwise the cost is zero, which means the inserted connector-edge is not part of the constrained interconnection pattern specified in the pattern-graph, hence inserting one or more new connector-edges does not violate the interconnection pattern.

Figures 5(c) to (i) illustrate several cases of matching imported connector-edges where the maximum cardinality is 2.

## 8 *BQ-$A^*$* search algorithm

We use the $A^*$ search algorithm that is modified by a "*bounded path-queue heuristic*" to compute a sub-optimal matching cost between the pattern-graph $G_i^p$ and input-graph $G_i^I$ while the AQL query constraints are not violated. The search algorithm generates a search-tree that corresponds to the recovery of each abstract-component $M_i$ in the query-graph $G^q$, that consists of: i) a *root tree-node* for matching the main-seed $n_j$ of the selected source-region $G_{g(i)}^{sr}$ with the first placeholder-node $n_{i,1}$ in the pattern-region $G_i^{pr}$; ii) a number of non-leaf tree-nodes at different *levels* of the search-tree that correspond to different alternative matching of the placeholders in the pattern-region with nodes in the source-region; and iii) leaf tree-nodes that correspond to solution paths where the placeholders have been matched and constrains have been met. At each node of a search-tree the cost of graph transformations for matching "a node $n_k$ and its edges" from the source-region with a "placeholder-node $n_{i,j}$ and its edges" from the pattern-region are evaluated and the search-tree is expanded from a tree-node that has the minimum cost. The cost evaluation was discussed earlier. Each search-tree has a maximal depth equal to the number of placeholder-nodes in the pattern-region (or equivalently to the maximum number of placeholders in the abstract-component $Mi$).

### 8.1 Tractability of the matching process

The proposed environment for architectural recovery in Figure 1 incorporates several techniques in order to tackle the inherent complexity of an architectural recovery processes and to provide a tractable and interactive recovery process. These techniques are discussed below:

**Sub-optimality to achieve performance**: A major drawback of the optimal search algorithms such as $A^*$ is the requirement to maintain all incomplete tree-paths (partially-matched graphs) in a sorted queue that allows to select the cheapest tree-path to expand next. This queue grows very fast and in the worst case can have an exponential size, which makes the process of storing and sorting the paths in the queue as a bottleneck for the algorithm. Since the path queue is sorted, all of the eligible paths to be expanded (i.e., low cost paths) are located toward the head of the queue. Therefore, most of the paths with high cost at the end of a large path-queue will never get a chance to be expanded, and remain at the tail of the path-queue until the end of a successful search. This property allows us to restrict the size of the path queue within a reasonable range (e.g., multiple hundreds of paths) at the expense of obtaining possibly a suboptimal solution. We call this algorithm *Bounded Queue $A^*$* (*BQ-$A^*$*).

**Search space reduction:** The whole search process is divided into a multi-phase search process, where at each phase the modified $A^*$ search (*BQ-$A^*$*) recovers an individual module using a reduced search space known as a source-region. Therefore, the whole search space, i.e., all nodes of the source-graph with the cardinality $n = |N^s|$ is reduced into $s = |N_{g(i)}^{sr}|$ nodes, which greatly contributes in relaxing the search complexity.

**Hierarchical architecture recovery:** The proposed architectural environment in Figure 1 allows to performs architectural recovery at two levels of granularity for the system entities, such that: i) at "file-level" a system of files is decomposed into a number of subsystems of files, and ii) at "function-level" each generated subsystem can be decomposed into a number of modules consists of functions, datatypes, and variables.

**Implementation considerations:** The implementation of the connector-edges is crucial in reducing the complexity of the proposed matching process. The main ideas are as follows: i) preventing highly repeated operations on the source/sink nodes of the connector-edges by caching the source/sink nodes; ii) simplifying the edge-bundle implementation by representing it as a positive integer. Therefore, deleting a whole edge-bundle or matching a part of edges in an edge-bundle simply means decrementing the integer by one; and redirecting an edge-bundle means no change on this value.

## 9 Incremental pattern generation & recovery

The architectural pattern that is defined in the AQL query and represented by the query-graph $G^q$ is generated through an incremental and interactive process, as described in the following steps:

**Step 1:** Decide on a method of main-seeds selection for the AQL abstract modules/subsystems. The proposed methods include: i) utilize knowledge about the related domain such as a reference architecture with well-defined components, design documentation, informal information embedded in the source-code, naming conventions, or directory structures; and ii) consider system analysis and metrics such as association structure of the system files and different methods of clustering [12]. The adopted method(s) should be able to suggest important and rather distinct main-seeds as the cores of functionality for the abstract module/subsystems in the AQL query. Generate an AQL query with zero modules/subsystems.

**Step 2:** Select the main-seed for the next module/subsystem and assign the number of placeholders, e.g., 10 for subsystem recovery and 20 for module recovery. Recover the new module/subsystem where no link constraints are defined for the new module/subsystem.

**Step 3:** Investigate the quality of the new recovered module/subsystem and its interaction with the already recovered modules/subsystems[2]. If the recovery is satisfactory go to Step 5. Otherwise, define (or adjust) the minimum/maximum link constraints between the new module/subsystem and one or more previous modules/subsystems, considering: i) increasing the maximum range causes the matching process to allocate higher scores to the group of entities that can augment the number of interactions to this maximum range; and ii) the minimum range is used to restrict the number of interaction to a minimum threshold, however it does not affect the scoring mechanism.

**Step 4:** Repeat the recovery process for the new module/subsystem with the constrained links. If the process is very lengthy due to backtracking, then interrupt the process and observe the tool-provided run-time information about the critical constrained links. Go to Step 3.

**Step 5:** If the number of the recovered modules/subsystems is not sufficient according to the user's preferences go to Step 2. Otherwise stop the recovery process and succeed.

If the number of remaining entities in the rest-of-system is high, an extra step, namely "constrained distribution" can be performed. In this step a part of the remaining entities in the rest-of-system are allocated to the recovered modules/subsystems based on the highest average closeness of each entity to one of the recovered modules/subsystems, provided that this allocation does not violate the link constraints. If a link constraint is violated the next highest module/subsystem is tried until the allocation to any of them violates the link constraints, where the entity is returned to the rest-of-system.

[2]The user's knowledge about the functionality of the modules/subsystems is required in order to impose meaningful constraints on the modules/subsystems interaction.

## 10 Case studies

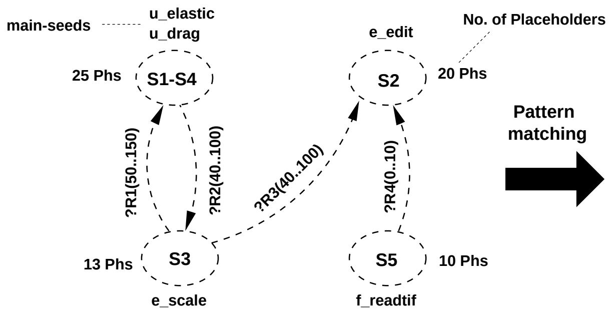
In this Section experimental results by applying the proposed approach are presented. A comprehensive set of experiments related to the time/space complexity, accuracy, stability, and quality of the architecture recovery technique has been presented in [12]. The proposed technique has been implemented in Alborz [11], a prototype toolkit that aims to recover the architecture of the medium size systems implemented in a procedural language such as C. The input to the Alborz tool is an information base that corresponds to the entities and relationships of the software system in the form of an AST or RSF file. The tool provides the result of the architectural recovery into two forms: i) HTML pages for the recovered components, tool generated metrics, and source code, to be visualized by a Web browser such as Netscape; and ii) graphs of boxes and arrows to be visualized by the Rigi tool [1], where the boxes are the system files or the analyzed components and the arrows are either the resource interaction (i.e., import/export) between the components or their association strengths. The association values among the system files are distributed over a wide range of values, hence they can be classified into several sub-ranges, namely "strengths of association" consisting of four sub-ranges of *strong*, *medium*, *loose*, and *weak*. This classification of values allows to simplify the visualization of the association graph of the system files or components (i.e., modules or subsystems).

The experiments are performed on six middle-size industrial systems, namely: i) *Xfig.3.2.3* drawing editor, ii) *Clips.4.20* expert system builder, iii) *Apache.1.2.4* http server; iv) *Bash.2.03* Unix shell; v) *Elm.2.5.6* Unix mail system; and vi) *Ghostview.3.5.8* postscript file viewer and navigator. Figure 6(a) presents the source-code related characteristics of the experimentation suite, however because of space limitation only one case is presented here.

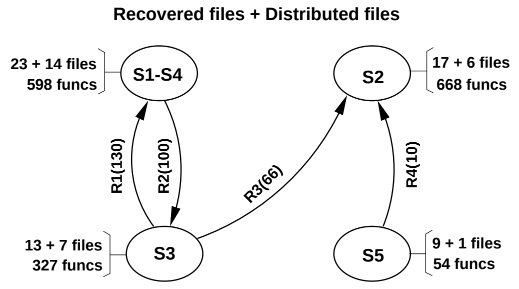
The hardware platform for the experiments consists of a Sun Ultra 10 with 440MHZ CPU, 256M memory, and 512M swap disk. The experiments are performed in a

| *Systems* | *KLOC* | *Files* | *Funcs* | *Aggr. types* | *Global vars* |
|---|---|---|---|---|---|
| Xfig | 74 | 98 | 1662 | 37 | 1356 |
| Clips | 40 | 44 | 736 | 54 | 161 |
| Apache | 38 | 42 | 709 | 42 | 95 |
| Bash | 44 | 47 | 1017 | 45 | 365 |
| Elm | 35 | 62 | 420 | 19 | 244 |
| GSview | 39 | 47 | 469 | 10 | 382 |

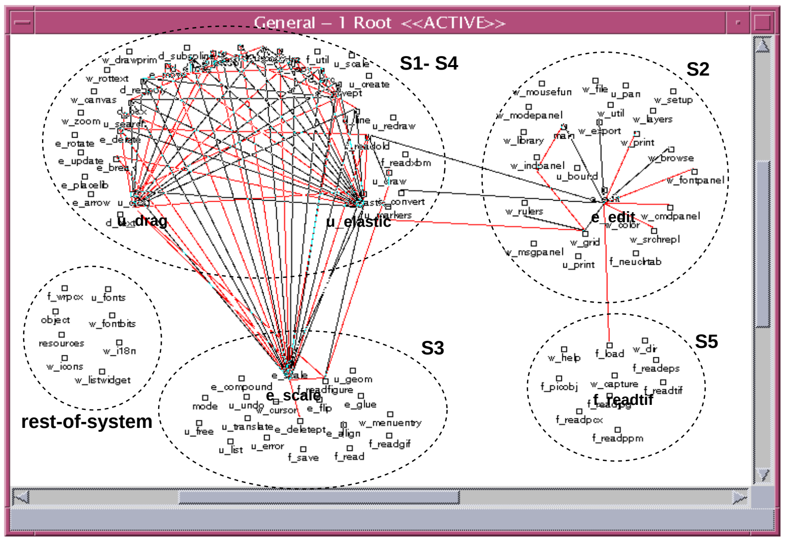
(a) Source code information of the six analyzed systems



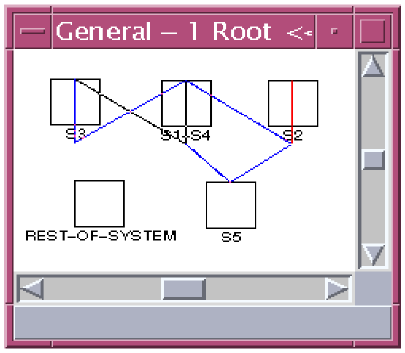
(b) Architectural pattern using AQL query



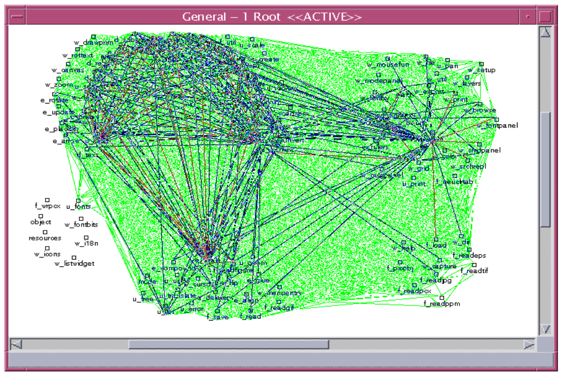
(c) Recovered architecture



(d) Final recovery of Xfig system: the subsystems S1 and S4 are merged into subsystem S1-S4



(e) Association links among the resulting subsystems in part (c).



(f) Adding all association links to part (b)

| Recovered subsystems | No. of files | Xfig subsystems | No. of files | Precision | Recall |
|---|---|---|---|---|---|
| S1-S4 | 37 | editing & utility & drawing | 47 | 81% | 63% 45% 100% |
| S2 | 23 | X-windowing | 28 | 78% | 64% |
| S3 | 20 | editing & utility & | 37 | 65% | 31% 39% |
| S5 | 10 | file manipulation | 16 | 70% | 44% |
| rest-of-sys | 8 | 5 zero size files | —— | —— | —— |
| Xfig subsystems: | | 1) editing: 19 files<br>2) utility: 18 files<br>3) drawing: 10 files | 4) file manipulation: 16 files<br>5) X-windowing: 28 files | | |

(g) Arcitectural recovery evalution

Figure 6. (a) Analyzed systems. (b) Architectural pattern of the Xfig system where the subsystems S1 and S4 have been merged. (c) Recovered architecture where the link constraints have been satisfied. (d) and (e) Graph visualization of the recovered subsystems for the Xfig system using the file association graph and subsystem association graph with "strong" and "medium" association strengths. (f) Viewing all association link strengths. (g) Architectural evaluation using "Precision" and "Recall" metrics.

single-user load environment.

**Architecture recovery of Xfig**: The Xfig system [2] lacks any documentation on its structure and only the user manual exists. However, a consistent naming convention is used throughout the system files which can be used as an aid for inferring its structure. Figure 6(b) illustrates the generated architectural pattern of the Xfig system with four abstract subsystems and corresponding link constraints according to the steps in Section 9. During the incremental and iterative recovery process this pattern yields the recovered architecture in Figure 6(c) where the size constraints for both the subsystems and links have been satisfied.

Figures 6(d) and (f) illustrate the file association graph feature of the proposed environment for viewing the Xfig recovered architecture. Figure 6(d) illustrates the result of the recovery process (only the strong and medium association links are shown) where the highly associated files are grouped into subsystem S1-S4 and the association among the subsystems are limited. Figures 6(f) illustrates the inclusion of the loose and weak association links to Figure 6(d). Figure 6(e) illustrates the association links among the recovered subsystems as a simplified view of the other figures. The subsystem S1-S4 has high association with subsystem S3 but low association with subsystems S2 and S5 as it was aimed for. Also in Figure 6(e) the lines across the boxes for the subsystems S1-S4, S2, and S3 indicate high intra-subsystem association that can be interpreted as the recovery of high cohesive subsystems. Figure 6(g) presents the accuracy of the Xfig recovery process in terms of the Precision and Recall metrics. The subsystem S1-S4 recovers all the drawing files and together with S3 recover almost all the editing and utility files. S2 is allocated to windowing files and S5 recovers file-manipulation files. The obtained Precision and Recall values indicate the accuracy for the proposed pattern matching technique.

## 11 Conclusion

This paper contributes to the reverse engineering research area by providing an interactive environment for architectural recovery, an incremental graph pattern matching model of the recovery process, and a prototype toolkit to support the proposed methodology. The proposed environment is based on techniques from the areas of data mining, approximate graph matching, clustering, and programming language design.

## References

[1] Rigi, URL = http://www.rigi.csc.uvic.ca/rigi/rigiindex.html.

[2] Xfig, URL = http://www.xfig.org/userman/.

[3] M. A. Eshera and K. S. Fu. A similarity measure between attributed relational graphs for image analysis. In *Seventh International Conference on Pattern Recognition*, pages 75–77, 1984.

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

[5] R. Holt, A. Winter, and A. Schurr. Gxl: Toward a standard exchange format. In *Proceedings of the WCRE'00*, pages 162–171, 2000.

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

[7] R. Kazman and S. J. Carriere. Playing detective: Reconstruction software architecture from available evidence. Technical Report CMU/SEI-97-TR-010, Carnegie Mellon University, 1997.

[8] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, 1997.

[9] S. Mancoridis, et. al. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the IWPC*, pages 45–53, 1998.

[10] 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, 1995.

[11] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *Proceedings of the IWPC'01*, pages 115–116, 2001.

[12] K. Sartipi. *Software Architecture Recovery based on Pattern Matching*. PhD thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2003.

[13] K. Sartipi and K. Kontogiannis. Component clustering based on maximal association. In *Proceedings of the IEEE WCRE'01*, pages 103–114, 2001.

[14] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the IEEE ICSM'01*, pages 408–419, Italy, 2001.

[15] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, Nov./Dec. 1999.

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