On Modeling Software Architecture Recovery as Graph Matching

Kamran Sartipi

Deptartment of Computing and Software
McMaster University
Canada
Sartipi@mcmaster.ca
http://www.cas.mcmaster.ca/~sartipi

September 25, 2003

Outline

- Motivation and definition for software architecture and software architecture recovery
- Issues to be addressed in a software architectural recovery environment
- Proposed approach to support reflective and patrern-based architectural recovery
- Conclusion and future research directions

Motivation for Software Architecture Recovery

- Average life-time of large systems is 10-15 years. Replacement of these systems is very expensive.
- Adopting a new technology such as: object-orientation, component-based programming, or network-centric requires changes in the design of system.
- Maintenance activities such as error-correction and feature enhancement, invalidate the design documents.
- Migrating a legacy system to a new platform such as Windows or Unix requires functional description of the system's components.

Software Architecture

- A generally accepted definition:

"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" [SEI 1994]

- However, software architecture is more than "components and connectors", or "major elements of a system". It is a collection of views, patterns, stakeholders, and roles [SEI].
- Therefore, Software architecture provides the necessary means to formalize and interpret the properties of a software system.

Software Architecture Recovery

Extracting high-level structural information from low-level system representation such as source-code

Major architecture recovery techniques:

- Clustering [MQ-partitioning, ACDC]
- Concept lattice analysis [Repairing, Horizontal]
- Pattern-based techniques [Dali, Recognizers]
- System visualization and analysis [Pbs, Rigi]

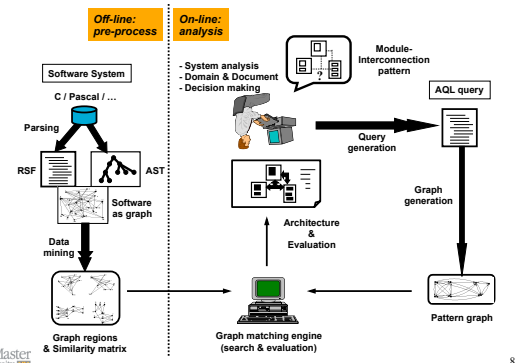
Issues to be addressed by an architectural recovery environment

- What view of the system to recover?
- How to represent the software system?
- How to model the high-level view of system?
- What recovery technique to use?
- How to scale the recovery process?
- How to involve the user in recovery?
- How to validate the architecture?

Graph Matching techniques

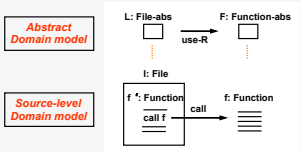
- Exact and approximate graph matching techniques:
  - Comparing primitives of prototype and input graph.
  - Decomposing the graphs into simple trees to match.
  - Generating an state space using cost of graph edit operations and search for minimum path.
- Graph in reverse engineering:
  - Adopted as standard for information exchange among tools.
  - Uniform mechanism for representing the software system and performing pattern matching process.

Environment for Pattern-based Software Architecture Recovery

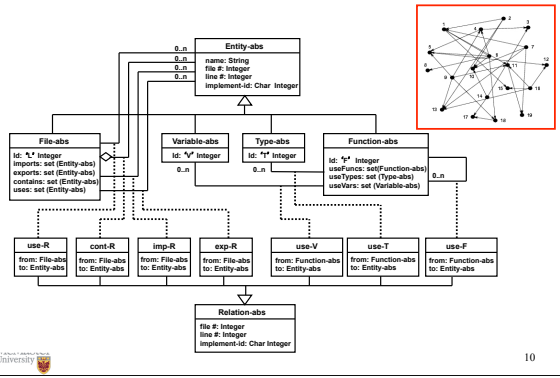


Software system representation

- Abstract domain model provides abstraction of the source-level domain model
- Entity-types: a subset of entity-types in source-code
- Relation-type: an aggregation of one or more relation-types in source-code

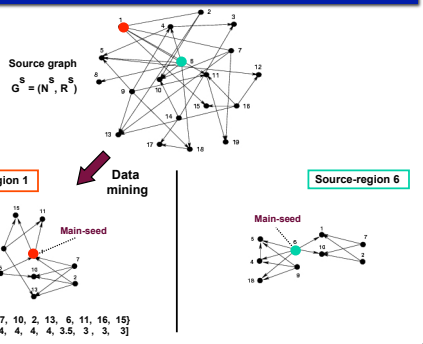


Domain model for software system



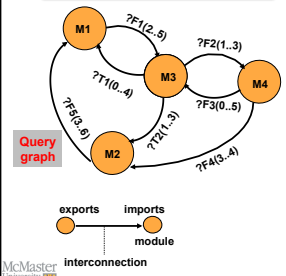
Dividing the system graph into regions

System representation: the collection of source-regions



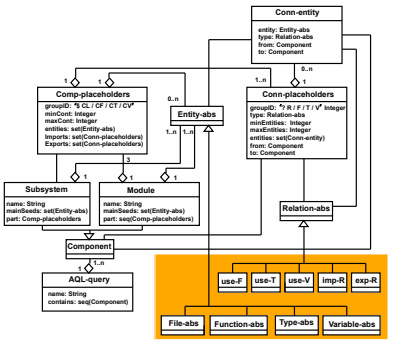
Modeling high-level view of system

Architecture Query Language (AQL)
Module Interconnection Pattern

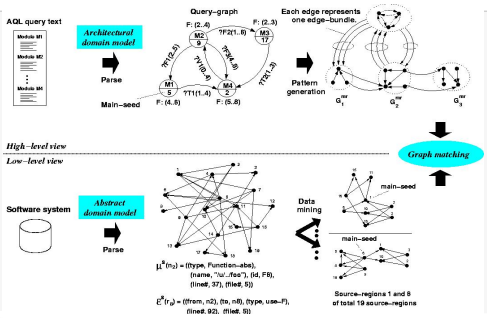



```
MODULE: M1
  MAIN-SEED:      func  search_class

  IMPORTS:
    FUNCTIONS:    func  ?IF,
                  func  ?F5(3..6)  M2
    TYPES:        type  ?IT,
                  type  ?T1(0..4)  M3
    VARIABLES:    var   ?IV

  EXPORTS:
    FUNCTIONS:    func  ?EF,
                  func  ?F1(2..5)  M3
    TYPES:        type  ?ET
    VARIABLES:    var   ?EV

  CONTAINS:
    FUNCTIONS:    func  $CF(15 .. 18),
                  func  search_class (),
                  func  inherit_facts (),

    TYPES:        type  $CT(0 .. 2)
    VARIABLES:    var   $CV(3 .. 5)

END-ENTITY
```

Domain model for AQL (Conceptual Architecture)



Graph Matching Model of Recovery



Approximate graph matching

- *f: G1 → G2* maps the nodes and edges of G1 onto G2.
- Different forms of function ***f***:
  - *Homomorphism*: ***f*** can map two nodes of G1 to one node of G2.
  - *Monomorphism*: ***f*** is one-to-one (i.e., sub-graph isomorphism).
  - *Isomorphism*: ***f*** is one-to-one in both directions.
- Exact graph matching:
  - Identifies exact set of nodes and edges of G1 that matches with G2 (in most real applications is not feasible).
- Approximate graph matching:
  - An optimal sequence of graph edit operations, such as: insertion / deletion of nodes and edges of G1 so that G1 and G2 become isomorphic.

Different types of graphs

- Source-graph: G^s
- Query graph: G^q
- Source-region: $G^{sr}_{g(i)}$
- Pattern-region: G^{pr}_i
- Input graph: G^I_i
- Pattern graph: G^p_i
- Matched graph: G^m_i

Modeling software architecture recovery as "graph pattern matching"

Given a query graph $G^q = (N^q , R^q)$ that is expanded to a pattern graph G^p,

given a system graph $G^s = (N^s , R^s)$, and

given a graph distance threshold d_t ,

the problem is to find a sub-graph of G^s i.e. G^m that approximately matches with the pattern graph G^p, so that:

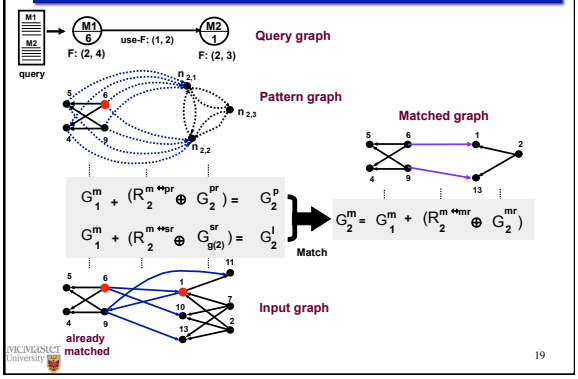
$$dist(G^p , G^m) < d_t \quad \& \quad dist(G^p , G^m)_{|min}$$

Graph algebraic model of matching process

$$G^m_{i-1} + (R^{m \leftrightarrow sr}_i \oplus G^{sr}_{g(i)}) = G^I_i$$
$$G^m_{i-1} + (R^{m \leftrightarrow pr}_i \oplus G^{pr}_i) = G^p_i \quad \Rightarrow \text{Match} \Rightarrow \quad G^m_i$$

- At each phase *i* of the matching process, G^I_i is approximately matched against G^p_i which results in G^m_i
- The graph edit operations are performed on *pattern-region* G^{pr}_i *and its edge-bundles* $R^{m \leftrightarrow pr}_i$ to match them against *selected source-region* $G^{sr}_{g(i)}$ *and its connector-edges* $R^{m \leftrightarrow sr}_i$

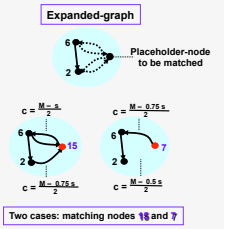
Example: incremental graph-pattern matching (phase 2)



Internal-edge deletion cost

- Objective: generating highly cohesive modules
- Internal-edge deletion cost must relate to:
  - **M**: maximal similarity between two nodes in the region
  - **s**: similarity between corresponding nodes
  - **k**: number of already matched nodes in the module
  - **d**: number of deleted edges between two nodes

$$c = \frac{M - s}{k} + \frac{0.25\ d\ s}{k}$$



Imported & exported connector-edge deletion costs

IMPORT

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

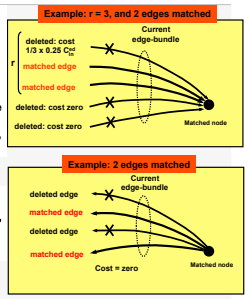
2- Keep **"r"** edges from the current edge-bundle and delete the rest with cost **"zero"**

3- Match the edges from **"r"** edges in edge-bundle

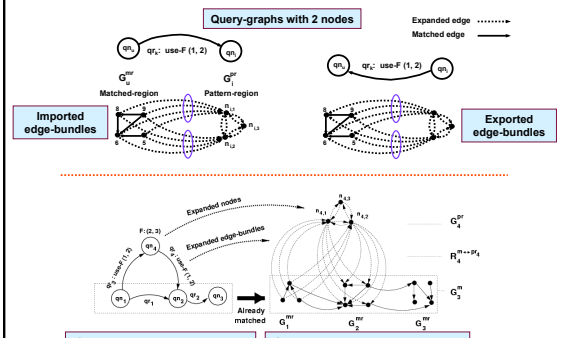
4- From **"r"** edges, each edge that is not matched, is deleted with cost:

EXPORT

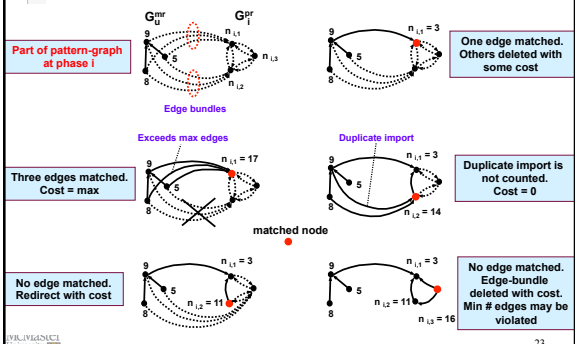
IF one or more edges matched from edge-bundle
THEN delete unmatched edges with cost **"zero"**
ELSE
delete all edges with cost: $0.25 \times C^{ed}_{in}$



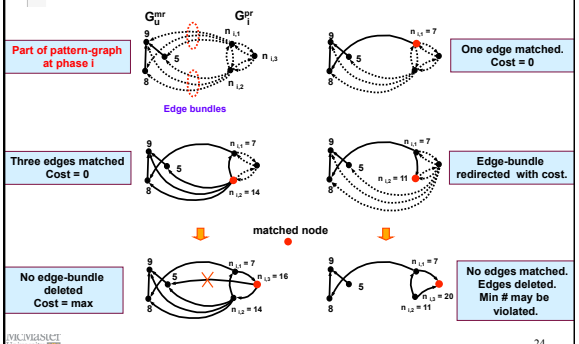
Generating pattern-graph from query-graph



Edge matching for imported edge-bundles



Edge matching for exported edge-bundle

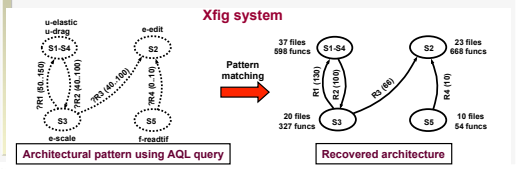


Steps for incremental pattern generation

1) Select main-seed for next module using tool provided techniques.

2) Recover next module with no link constraints

3) Based on the interaction with other components, and user's objectives define the constrained links for this module.
   * Maximum range is used to encourage high interaction
   * Minimum range is used to restrict the number of interaction



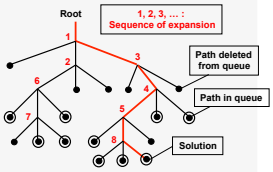
Techniques to Address Tractability

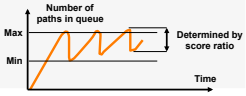
- Incremental recovery by dividing the search space into sub-spaces
- Hierarchical recovery
  - Decomposing system into subsystem of files
  - Decomposing a subsystem into modules of F/T/V
- Sub-optimal search techniques, e.g., bounded path-queue A* (BQ-A*)
- Implementation techniques

A^* search with Bounded path-queue

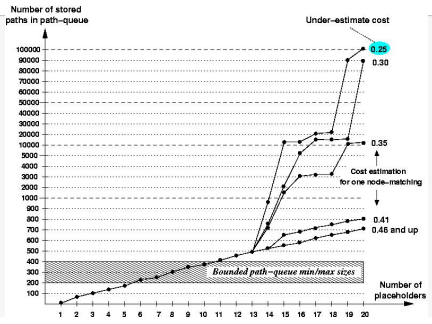
Sub-optimal solution to achieve tractable search.

- A^* produces queue of sorted incomplete paths.
- Storing, sorting, duplicate path checking are bottlenecks.
- In successful search most of paths at the end of queue are not expanded.
- Max / min thresholds: multiples of the size of domains.





Space Complexity Reduction



Implementation Related Techniques

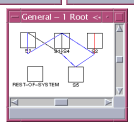
- HERE THE WAYS THAT THE COMPLEXITY REDUCDES:
- HOW PRESENT THE EDGE-BUNDLES
- CACHING THE INFORMATION OF THE SOURCE AND SINK NODES OF EDGES
- EXPONENTIAL COMPLEXITY WHEN SEARCH SPACE IS REDUCED TO SOURCE-REGIONS

User assistance

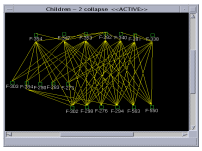
- Statistical Metrics
  - Overall association among files
  - Fan-in fan-out
  - Design views
- Visualization
  - Simplifying the graph views
  - Browsing mechanism through HTML pages
- Assistance with pattern generation
  - Identifying the locus of interactions

Representing the architecture using graph visualizer (Rigi)

- Different types of links between boxes:
  - Association-links
  - Entity-usage links
- Association-links with different strengths to simplify the view
- Viewing the locus of interaction among entities to evaluate the recovery process
- Insight into the system before starting the recovery
- Manual recovery

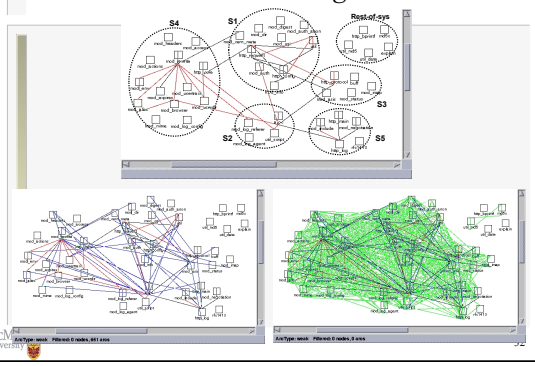


File-level analysis



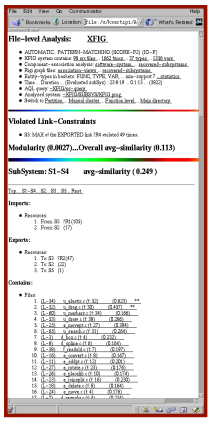
Function-level analysis

Architecture of Apache 1.2.4 Partitioning



Representing the architecture using Web browser (NetScape)

- Hypertext links to actual entities in the source file.
- Information presented includes:
  - Evaluation metrics: modularity quality, average similarity
  - Statistical information for link-constraint violations
  - Interactions among components
  - Browsing the query
  - Switch between file-level and function-level analysis



Validation of the recovery

- Modularity quality
  - Connectivity based
  - Association based
- User investigation of the graphs
  - Simplified graphs
- Conformance with documented architecture
  - Precision and Recall

Accuracy of the recovered architecture

- Clips expert system
  - 40 KLOC
  - 44 files
  - 736 functions
  - 161 global vars
  - 54 aggregate types
- Xfig drawing editor
  - 74 KLOC
  - 98 files
  - 1662 functions
  - 1356 global vars
  - 37 aggregate types

Recovered subsystems	No. of files	Clips subsystems	No. of files	Precision	Recall
S1	11	- Defrule structures - Inference engine	13	82%	70%
S2	10	- Rule manipulation	6	50%	83%
S3	4	- Object	3	75%	100%
S4	4	- Expression eval	4	75%	75%
S5	10	-System function -User interface	7	49%	57%
rest-of-sys	5	————	——	———	——

Recovered subsystems	No. of files	Xfig subsystems	No. of files	Precision	Recall
S1-S4	37	editing & utility & drawing	47	81%	63% e- 46% u- 100% d-
S2	23	X-windowing	28	78%	64% w-
S3	20	editing & utility	37	65%	31% e- 39% u-
S5	10	file manipulation	16	70%	44% f-
rest-of-sys	8	6 zero size files	——	———	——

Conclusion

- Presented an interactive environment for architectural recovery and evaluation, and the supporting toolkit
- Highlights of the approach:
  - Modeled the recovery process as “graph pattern matching”
  - Used data mining techniques to define similarity metric
  - Limited the complexity of recovery process by two techniques
  - Developed a query language based on ADL features
  - Represented the recovery result through HTML pages and graphs to be visualized

Future directions

- Behavior recovery:
  - Extracting frequently repeated traces of event using techniques such as "sequential pattern discovery"
- Recovery of more architectural styles
  - Pipe & filter
  - Client & Server
- Conformance with standard information exchange GXL

A Pattern-based Environment for Architectural Recovery and Evaluation

Kamran Sartipi

Software Engineering Group
School of Computer Science
University of Waterloo

Ksartipi@matah.uwaterloo.ca
http://swen.uwaterloo.ca/~ksartipi

May 29, 2003

Web services

- Systems integration requires more than the ability to conduct simple interactions by using standard protocols
- The full potential of Web Services as an integration platform will be achieved only when applications and business processes are able to integrate their complex interactions by using a standard process integration model.
- Models for business interactions typically assume sequences of peer-to-peer message exchanges. Both synchronous and asynchronous, within stateful, long running interactions involving two or more parties

Motivation

- Pattern matching problem
- Clustering problem
- Constraint satisfaction problem
- Lattice partitioning problem
- Composition and visualization problem

Motivation
Lack of a reflective and uniform model for pattern-based architectural recovery, whereby the software system, architectural pattern, and pattern matching process, are all uniformly represented using a graph formalism.

System representation: "Attributed Relational Graph (ARG)"

An "ARG" is a six-tuple G = (N, R, A, E, f, g):

- N = {n_1, n_2, …, n_n}: attributed nodes (entities)
- R = {r_1, r_2, …, r_m}: directed attributed edges (relations)
- A & E: alphabets for node & edge attributes/values
- **μ** & **ε**: node & edge labeling functions

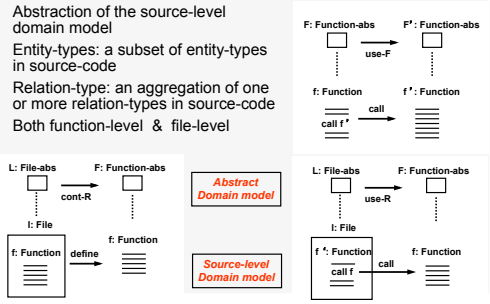
Example of attributes in software system:

- Label: path-name and identifier for nodes and edges
- Type: type of node or edge
- Location: two integers for *file#* and *line#*

μ(n_2) = (type, Function-abs), (name, "/u/…/foo"), (id, F6)
ε(r_8) = (from, n_2), (to, n_{34}), (type, use-F), (line#, 92), (file#, 5))

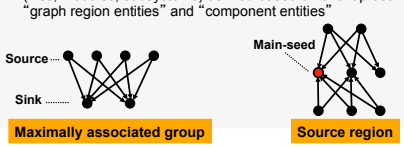
Abstract Domain Model

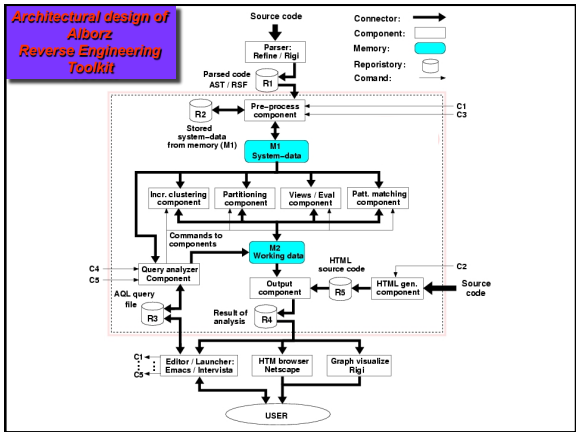
- Abstraction of the source-level domain model
- Entity-types: a subset of entity-types in source-code
- Relation-type: an aggregation of one or more relation-types in source-code
- Both function-level & file-level



Similarity between two entities based on maximally associated groups

- Maximally associated group:
  - A maximum group of entities (sources) sharing the same relations on another maximum group of entities (sinks)
- Source region:
  - Collection of entities that are associated with a region's main-seed
- Similarity between two entities:
  - Defined based on source and sink nodes in an associated group
- Similarity between two components:
  - (files, modules, subsystems) defined based on overlap between "graph region entities" and "component entities"





Software Architecture Recovery

Extracting high-level structural information from low-level software representation (e.g., source code)

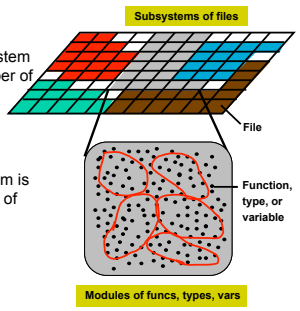
- Consists of two phases:
  - Extraction phase: an automatic tool generates source-model.
  - Analysis phase: a user-assisted tool constructs architecture.
- Constitutes a major part of software maintenace.
- Should relate with specific re-engineering objectives

Employed recovery techniques

- Subsystem containment hierarchy to achieve
- Concept lattice analysis
- Data mining techniques
- Pattern based techniques (graph matching) ✓
- Clustering techniques
  - Hierarchical
  - Partitioning ✓
  - Incremental ✓

Considering different levels of abstraction

- At file-level the software system is decomposed into a number of subsystems of files
- At function-level a subsystem is decomposed into a number of modules of functions, datatypes, and variables



Graph distance

Min-cost of a number of changes that are performed on graph G1 to transform it into graph G2

- Specific characteristics:
  - No node / edge relabeling
  - No node insertion since maximum size of nodes expanded
  - Node deletion is allowed up to minimum size
- A certain cost is associated with each graph change:
  - Connector-edge deletion / insertion cost to comply with pattern
  - Internal-edge deletion cost to achieve cohesive modules

McMaster University

49