Software Architecture Recovery based on Pattern Matching

by

Kamran Sartipi

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2003

Abstract

Pattern matching approaches in reverse engineering aim to incorporate domain knowledge and system documentation in the software architecture extraction process, hence provide a user/tool collaborative environment for architectural design recovery. This thesis presents a model and an environment for recovering the high level design of legacy software systems based on user defined architectural patterns and graph matching techniques.

In the proposed model, a high-level view of a software system in terms of the system components and their interactions is represented as a query, using a description language. A query is mapped onto a pattern-graph, where a module and its interactions with other modules are represented as a group of graph-nodes and a group of graph-edges, respectively. Interaction constraints can be modeled by the description language as a part of the query. Such a pattern-graph is applied against an entity-relation graph that represents the information extracted from the source code of the software system. An approximate graph matching process performs a series of graph edit operations (i.e., node/edge insertion/deletion) on the pattern-graph and uses a ranking mechanism based on data mining association to obtain a sub-optimal solution. The obtained solution corresponds to an extracted architecture that complies with the given query.

An interactive prototype toolkit implemented as part of this thesis provides an environment for architecture recovery in two levels. First the system is decomposed into a number of subsystems of files. Second each subsystem can be decomposed into a number of modules of functions, datatypes, and variables.

Acknowledgment

There are several individuals who made contributions to this work. I would like to express my sincere appreciation to Prof. Kostas Kontogiannis who exposed me to the field of reverse engineering and supervised this thesis. He spent countless hours discussing different aspects of my research and his friendly attitude and scientific support are admirable. I would also like to thank Prof. Farhad Mavaddat for his guidance and support. Special thanks to my external examiner Prof. Panos Linos for carefully reading this thesis and providing constructive criticism, as well as for traveling to Waterloo. Also, special gratitude to Prof. Paulo Alencar for reading the early drafts of my thesis and providing valuable comments to improve it. I appreciate Professors Michael Godfrey and Sagar Naik for participating in my PhD committee and making precious suggestions.

I learned many lessons throughout the years of discussion, encouragement, and debate with my friends in the School of Computer Science and Department of Electrical and Computer Engineering. I was also blessed to have a warm community of Iranian friends in Waterloo during my studies. I consider this as my most valuable asset which provided me and my family an enjoyable environment in Waterloo.

This thesis is offered to my beloved family Noushin, Melody, and Ramona whose love and emotional support were always the driving force for me. Also, the love and blessing of my father Hojatollah, my mother Soghra, and the support of my brother Dr. Kambiz Sartipi gave me energy and endurance throughout this unforgettable journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For several decades we have been witnessing the impact of large software systems on our every day life. Legacy software systems are mission critical, large, and complex systems that are operational for approximately 10 to 15 years [110]. Due to prolonged maintenance such legacy systems are difficult to maintain, evolve, or integrate and in most cases their architectural design deviates from the original design. In this context, architectural recovery is a key activity in supporting maintenance tasks such as re-engineering, objectification, or restructuring.

In a nutshell, the approaches to software architectural recovery can be classified as clustering-based techniques and pattern-based techniques. The clustering-based techniques generate architectural components by gradually grouping the related system entities using a similarity measure [63, 112, 15, 107]. On the other hand, the pattern-based techniques first compose a high-level mental model of the system architecture (also known as *conceptual architecture* or *architectural pattern*) using a modeling means such as a query language [56, 55, 79, 48, 39] or a block diagram

[38, 78], and then a pattern matching engine searches to identify an instance of the architectural pattern in the software system.

1.1 Software architecture

There is no standard, universally-accepted definition for the term *Software architecture.* A number of researchers have attempted to provide a precise definition for software architecture and the Software Engineering Institute (SEI) has provided a collection of these definitions in [10]. In this Section, first two generic definitions of software architecture are presented, and then a definition of software architecture within the scope of this thesis is provided.

A classic definition by Garlan and Shaw [97] is as follows:

"software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns."

In this definition, a software architecture is identified by its components and connectors (as elements) and an style that are defined below.

A *component* represents the encapsulation of the system's computation corresponding to a compilation unit of programming languages. A component's interface defines the signature and functionality of the services provided by the component. Examples of components include: filter, process, layer, client, server, memory [97].

A *connector* represents the encapsulation of the interactions among two components. The connector protocol defines the form of communication, the type and order of the transported data, and the way of assuring the protocol execution.

Examples of the connectors include: RPC, event broadcast, and pipe [97].

An *architectural style* (also known as architectural pattern) captures the broad properties, vocabulary, and configuration constraints that apply to all instances of a family of systems. [77]. A collection of architectural styles, defined in [44], include: *pipe and filter*, *client and server*, *implicit invocation*, *layered, blackboard, object-oriented, interpreter, state transition*, *module and interconnection*, and *main program and procedure* [97].

An alternative definition for software architecture that is used in Nortel's Software Engineering Analysis Lab. is presented below. This definition is adapted from [42], focusing on the evaluation properties of a software architecture and the role of stakeholders:

"software architecture consists of: a collection of software and system components, connections, and constraints; a collection of system stake-holders' needs statements; and a rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stake-holders' needs statements."

In this thesis the software architecture, component, connector, and architectural pattern are defined as follows.

Software architecture: a partition of the software system entities into cohesive components that reflect the system characteristics and domain knowledge, and meets the structural constraints defined by a given architectural pattern.

Component: a named grouping of system entities (e.g., files, functions, datatypes, and variables) that interacts with other groups through using or providing the required system entities (e.g., functions, datatypes, and variables).

Connector: defined between two components as a group of system entities (e.g., functions, datatypes, and variables) that are defined in the source component and are used by the destination component.

Architectural pattern: a set of partially specified components and a number of (size and type) constrained connectors among the components that collectively represent a macroscopic view of the core functionalities and interactions within the software system.

Therefore, the architectural pattern in this thesis can be categorized as a "module and interconnection" pattern. For simplicity an "architectural pattern" is also referred to as a "pattern".

1.2 Software architecture recovery

Software architectural recovery encompasses various methods for extracting architectural information from a lower level representation of a software system such as the source code. The whole architecture recovery process is divided into two phases. In the first phase, namely the *extraction* phase, a tool automatically builds a more abstract system representation, i.e., the *source model*, out of the program representation. In the second phase, namely the *analysis* phase, a user-assisted tool constructs a high-level view of the system from the source model. Most approaches to software architecture recovery view the recovery process either 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 [56, 79, 55, 48]; ii) a clustering problem that models the recovery by grouping the related parts of a software system into cohesive components [63, 78, 59, 72, 107]; iii) a constraint satisfaction problem that models the recovery by identifying groups of entities that meet the conditions defined in a repository of plans [113]; iv) a lattice partitioning problem that models the recovery by classifying maximally related groups of entities that are arranged in a lattice [98, 68, 109]; or v) a composition and visualization problem that models the recovery by aggregating system entities into containment-hierarchy of components [78, 38, 69].

1.3 Motivation

The reverse engineering community has paid particular attention to the pattern matching approaches since they allow the use of domain knowledge and system documents in composing the architectural 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 [19, 49, 23].

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.

1.4 Thesis hypotheses and problem description

In this Section, we consider four hypotheses that are used to formulate the thesis in this dissertation and also to define a coherent problem description. Then, the corresponding issues that must be addressed by an approach to software architecture recovery, are discussed.

Hypothesis 1: it is generally accepted that the software architecture recovery should be performed with respect to specific objectives, such as increasing understandability, maintainability, or adaptability of the software system. Also the result of the recovery must be testable to indicate the degree of success in achieving such objectives.

Hypothesis 2: the design decisions governing the allocation of software system's functionality onto its structure mainly originate from the knowledge within the software system's application domain. Such design decisions are considered as an important part of the design documentation of the system. Both types of information significantly assist in the recovery of the system's structure.

Hypothesis 3: the clustering-based approaches to architectural recovery can only implicitly guide the result of the recovery toward a constrained objective. In contrast, the pattern-based approaches can use an expressive formalism to model structural constrains that are derived from the application domain and system documentation and can be used to link the analysis results with intended objectives.

Hypothesis 4: the pattern-based architecture recovery approaches suffer from the lack of a semantically rich and expressive representation of the architectural patterns, and in most cases an expert user is required to formulate the architectural pattern.

Thesis: this dissertation argues that a pattern-based environment for software architecture recovery with an expressive architectural pattern language that incorporates knowledge from the system's domain and documentation, and a process that ensures a repeatable recovery result would best suit to the requirements of the architectural recovery mission.

Specifically, this thesis proposes a pattern-based architecture recovery approach whose objectives can be specified in terms of the structural properties that are defined through an architectural pattern. The proposed architectural pattern is defined based on the expressive features of the architecture description languages (ADLs) and is incrementally generated via an interactive procedure that allows to incorporate the knowledge from the application domain and system documentation. The result of the recovery can be directly tested against the recovery objectives through: i) conformance checking with the available documentation that ensures the decomposition of the core system functionality into components; ii) measuring the modularity quality of the recovered architecture to ensure the recovery of a maintainable system; and iii) conformance with the component and connector size and type constraints imposed by the pattern.

Problem description: the following observations are the basis for the problem definition in this thesis. 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 impossible to define the architecture of a large system at once, hence, the architectural recovery should be an incremental process. Software systems usually consist of architectural 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 thesis defines software architectural recovery problem as:

> *devising a tractable process, required techniques, and supporting tools for interactively and incrementally extracting a system's structure using domain and system knowledge.*

Specifically, an approach to software architecture recovery must address the following issues.

1.4.1 Views of the system to recover

The views of a software system are the result of applying *separation of concerns* on a development or reverse-engineering process of the software in order to classify the related knowledge about that process into more understandable and manageable forms. Unfortunately, reverse engineering is much more difficult to achieve than forward engineering. Recovering the functionality of a large and poorly documented legacy system is a non-trivial, if not impossible, task. In this respect, the structural view is the most tractable architectural view to recover, and a number of approaches and tools [56, 79, 38, 106, 98, 68, 72, 1, 116] already exist to address this issue in various ways. Other views to be considered are behavior [28] and data [51, 22].

1.4.2 Representation of the software system

In software architecture recovery an appropriate representation of the software system is important in both extracting the desired properties from the software, and providing support for programming language independent analysis. In general, the preserved information and the level of abstraction for analysis are trade-offs that need to be considered at this stage. The source code of the system being analyzed is parsed or scanned and an annotated abstract syntax tree (AST) [33, 39, 48, 45] or an entity-relation model such as GXL [49], TA [38] or RSF [1] can be derived.

1.4.3 Modeling high-level representation

The software architecture recovery is mostly viewed as a top-down process where the high-level view of the system (also known as the conceptual architecture or *architectural pattern*) is built as the user's mental model of the system architecture. The model serves as an hypothesis and should be able to represent an abstraction of components and their interactions as well as, a mechanism to constrain the type of such system entities and data/control dependencies. The existing approaches use either proprietary languages to model high-level architectural pattern of the system [79, 48, 39, 55], SQL queries to model the system that is to be validated against the source code representation [56], or box and arrow diagrams to visualize a subsystem view of the application being analyzed [38, 78].

1.4.4 Tractability of the recovery process

Searching for a particular property or groups of related properties in a large data base is a computationally intensive process. In some cases, the search algorithms are intractable for a large number of inputs, for example finding a subgraph pattern in a graph representation of a large system. Efficient techniques and heuristics are essential in managing the inherent complexity of architectural recovery tasks [55, 114]. In the case of very large systems, strategies such as considering naming conventions for files, using directory structure, and producing a containment hierarchy structure are often used [24].

1.4.5 User assistance

The role of the user, as an integral part of an architectural recovery process, is important for the validation of the obtained results. In fact, the ambitious goal of fully automating the recovery process is no longer supported by the research community. Instead, a cooperative environment of human and tool is the most promising solution for relaxing the recovery complexity [56, 27, 38, 79]. This trend necessitates that the domain knowledge and system documents be incorporated in the recovery process by the user involvement. In such a cooperative environment, the mission of the tools has also been shifted from complex search and recovery strategies to semi-automatic, user assisted based strategies allowing a variety of domain-specific information to be considered during the recovery process [38, 27]. In this context, the new terms such as *librarian* and *patron* [38] refer to the system information accumulation for human usage.

1.4.6 Validation of the recovered architecture

Similar to a validation test in forward engineering, a reverse engineering process is expected to prove the conformance of the recovered high-level hypothesis specification with the actual architecture of the software system. However, the validation of a recovered architecture is still in its early stages and requires more attention from the research community. In most cases, validation is performed by the user who evaluates the conformance of the obtained results with the system's architectural documents (if available and updated). Moreover, when the architectural documents and domain knowledge are available the Precision and Recall metrics (i.e., measuring the degree of conformance between the recovered architecture and the documented architecture) from the information retrieval domain can be used as a reliable objective measure. Recently, some approaches proposed other objective measures for validation of the recovery result [76, 60, 64].

1.5 Scope of the proposed solution

As part of this work, a survey on a number of industrial and experimental architecture description languages and reverse engineering tools[1] has been conducted. As a result, a collection of essential features used for describing software architectures has been identified and categorized into three architectural views that we believe are suitable for architectural recovery purposes. Figure 1.1 illustrates the proposed views where the grey region identifies the scope of recovery approach to be presented in this thesis. The set of views consist of *structure view, behavior view*, and *surrounding view* which are almost orthogonal and carry most of the important information about the software systems to be recovered.

The features and views in this classification encompass the systems in different domains such as information, concurrent, reactive, distributed, and command and control. The identified views of a software system for recovery are discussed bellow.

Structure view: *refers to the building blocks and interconnections (glues) that statically describe the architecture of a software system.* Structure view consists of two parts:

- *Static* features are the property of the source code, hence can be extracted by statically analyzing the source program. The static features are divided into three parts: i) an *entity* refers to a language construct, as a basic block, that constitutes in building a software's structure; ii) a

[1]The survey was conducted on the ADLs: Unicon [96], Rapide [70, 71], ACME [43], ARDEC DSSA [103], Wright [14], ControlH [20], PBS [38, 2], Polylith [82], Genoa [33], and DECODE [83]. A more detailed discussion on these ADL's is presented in the Section 2.3.

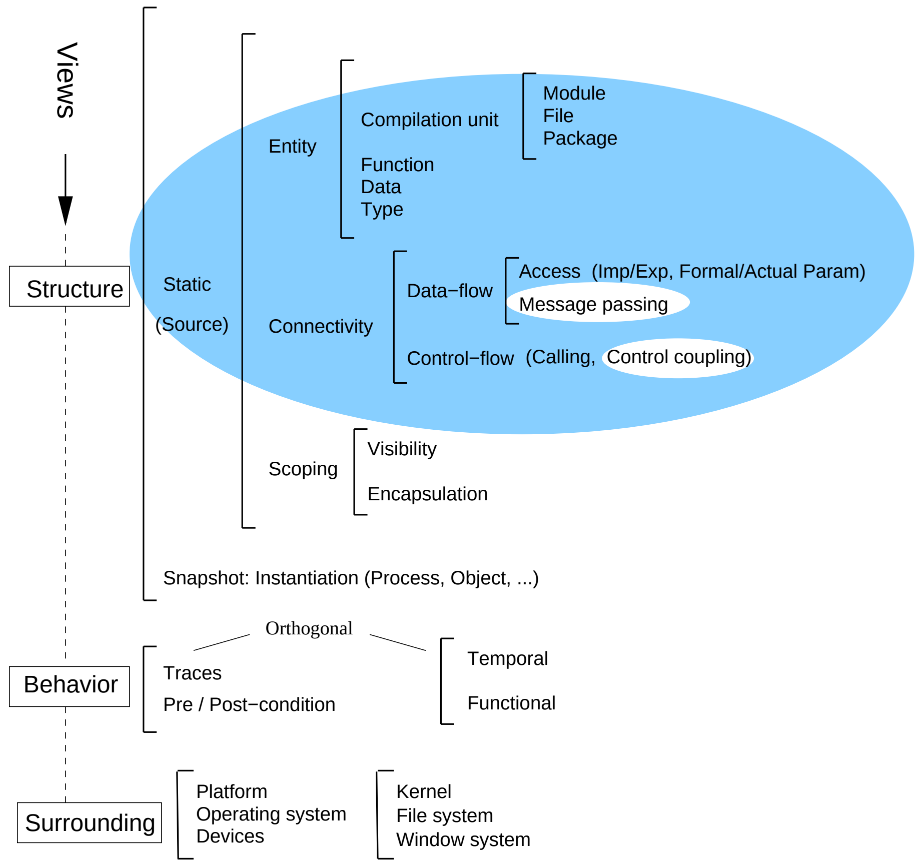


Figure 1.1: The set of architectural views and system features to be used for architectural recovery. The grey region highlights the scope of our approach to software architecture recovery.

connectivity refers to an interconnection between two entities; iii) a *scope* refers to the maximum range that a definition is effective.

- *Snapshot* features change over time, hence represent dynamic aspects of a program. These features can be detected statically by interrupting a running program and registering the program's context and state. Spawned concurrent processes, class instances (objects), etc. are typical pieces of information to be discovered.

Behavior view: *refers to the services that a system provides through its interactions with the surrounding systems.* Behavior view can be regarded as two orthogonal set of features:

- In one aspect, the system's behavior can be expressed by *event traces* and *pre/post-conditions* which is more appropriate for inspection by an analyst. Event traces represent the state transition model of a system. Pre/post-condition predicates specify the input/output constraints of a system's function [47].
- In another aspect, the system's behavior can be expressed by its *temporal* and *functional* properties which are difficult to be analyzed and recovered. The temporal properties of the system's behavior (also known as dynamic, run-time, and execution) refer to time-dependent characteristics, such as concurrency, synchronization, and communication. These properties are often too complex to be recovered. The functional properties of the system's behavior refer to data transformation characteristics. The algorithmic aspects of transformation such as computational complexity and memory usage are also important issues.

Surrounding view: *refers to all supporting facilities, including hardware/software and interface, that encompass the software system (an application program) and enable it to operate and provide its services to its interfacing systems.*

1.6 Proposed solution

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 given user-defined architectural pattern.

1.6.1 Environment for architecture recovery

Figure 1.2(a) illustrates 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. Also the computational expensive operations are highlighted. The proposed architectural recovery environment consists of two phases. During the *off-line pre-process* phase the required architectural information are extracted from the software system and are stored in a database for further use. This phase is usually time-consuming, however, it is performed once for each software system.

During the *on-line analysis* phase, the user specifies the architectural pattern of the software system and uses an interactive and iterative recovery process to recover the system architecture in terms of components and interactions that conform with the constraints of the architectural pattern. The proposed environment employs techniques from data mining in the off-line phase and techniques from approximate

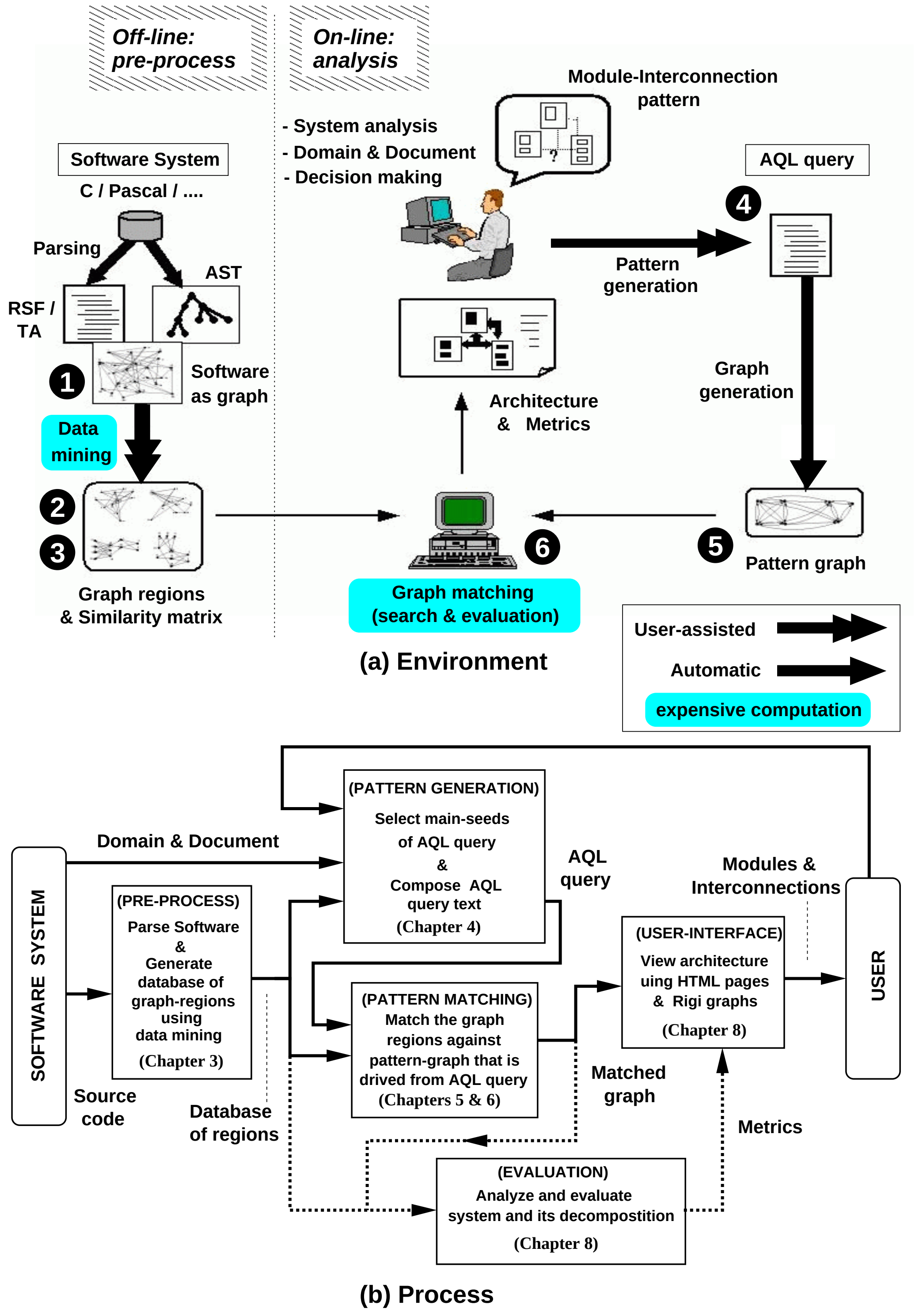

Off-line: pre-process
On-line: analysis
Module-Interconnection pattern
- System analysis
- Domain & Document
- Decision making
Software System
C / Pascal / ....
AQL query
Parsing
AST
RSF / TA
Pattern generation
Software as graph
Graph generation
Architecture & Metrics
Data mining
Graph regions & Similarity matrix
Graph matching (search & evaluation)
Pattern graph
(a) Environment
User-assisted
Automatic
expensive computation
SOFTWARE SYSTEM
Domain & Document
(PATTERN GENERATION) Select main-seeds of AQL query & Compose AQL query text (Chapter 4)
AQL query
Modules & Interconnections
(PRE-PROCESS) Parse Software & Generate database of graph-regions using data mining (Chapter 3)
(USER-INTERFACE) View architecture uing HTML pages & Rigi graphs (Chapter 8)
USER
(PATTERN MATCHING) Match the graph regions against pattern-graph that is drived from AQL query (Chapters 5 & 6)
Source code
Database of regions
Matched graph
Metrics
(EVALUATION) Analyze and evaluate system and its decompostition (Chapter 8)
(b) Process

Figure 1.2: The interactive environment and process for the proposed pattern-based software architecture recovery. The numbered solid circles locate the places where the thesis has contributions.

graph matching, clustering, and architecture description languages in the on-line analysis phase. In Figure 1.2(a), the numbered solid circles indicate the contributions of the thesis in different parts of the environment. In the following Section, the architectural recovery process for the proposed environment is discussed.

1.6.2 Process for architecture recovery

Figure 1.2(b) illustrates the architectural recovery process which serves as a complement to the information provided by the environment in Figure 1.2(a). The recovery process diagram indicates the techniques used in different stages and provides references to the corresponding Chapters of the thesis. The steps for off-line and on-line phases are as follows.

Off-line phase

Step 1, parsing: the software system, written in a procedural language such as C, is parsed and presented as a graph whose nodes and edges conform with a domain model that is suitable for architectural recovery and provides programming language independence for the recovery process.

Step 2, information extraction: the graph representation of the software system is further processed using data mining techniques, and consequently it is divided into a collection of subgraphs (as graph regions) where the appropriate graph regions 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.

On-line phase

The on-line analysis phase consists of three steps that are iteratively performed in a loop until the user stops the process based on the inspection of the quality of the recovered components, or the amount of overlap between the recovered components. The architectural recovery can be performed at two levels of abstraction. At the *file-level*, the software system is partitioned 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. At each iteration of the on-line phase the following steps are performed.

Step 3, pattern generation: in the first iteration of the on-line phase, the user specifies an architectural pattern that consists of only one component (module or subsystem). In the subsequent iterations, the user augments a constrained architectural pattern of the system components and their interactions as defined in Section 1.1, based on: domain knowledge, system documentation, or tool-provided system analysis information. This architectural pattern is defined using a proprietary language that we call *Architecture Query Language* (AQL).

Step 4, pattern matching: the approximate pattern matching engine generates a pattern-graph from the user-defined architectural pattern defined as an AQL query, and performs a sub-optimal match between the pattern-graph and the software system graph regions stored in the database during the off-line phase. The result of the matching process is a partially recovered architecture for the current iteration.

Step 5, result evaluation: using the means provided by the environment, such as: i) metrics for measuring the modularity quality of the partially recovered architecture; ii) inspection of the overlap between the currently recovered component with previous components; and iii) visualization of the interactions or association among the recovered components, the user may decide to proceed with a new iteration by branching to Step 3, or stop the recovery process.

1.7 Thesis contributions

This thesis presents an environment for software architecture recovery that employs techniques from approximate graph pattern matching, data mining, clustering, and architecture description languages. Specifically, the major contribution of this thesis is:

an environment for modeling software architecture recovery as graph pattern matching problem.

where the conceptual architecture of the system is formulated using a query-graph whose expansion is matched by the system entities and relationships through an incremental and iterative approximate graph matching process. Specifically, the contributions of the thesis with respect to the solid circles in the environment of Figure 1.2(a) are as follows:

Circle 1: A new domain model that allows to represent the software system as an attributed relational graph at a higher-level of abstraction than the source-code which is suitable for architectural recovery (presented in Chapter 3).

Circle 2: Two new similarity metrics that are based on the structural properties of the groups of entities with maximal association generated by data mining techniques. These similarity metrics are defined between two system entities and between two groups of system entities and are used to recover cohesive components through pattern matching process (presented in Chapter 3).

Circle 3: A novel technique to limit the computational complexity of the graph matching process for architectural recovery using graph regions (presented in Chapter 3).

Circle 4: An architecture query language (AQL) to model an architectural pattern of components and their interactions (presented in Chapter 4).

Circles 5 and 6: A new approximate graph matching algorithm to match a pattern-graph that represents a modular architectural pattern of the software system with a graph that represents a software system entities and their data and control dependencies (presented in Chapters 5 and 6).

As a part of this work, a toolkit (called *Alborz*) to perform interactive and incremental architectural recovery and evaluation has been implemented that provides the recovery environment illustrated in Figure 1.2(a). The toolkit and related experimentations are discussed in Chapter 8.

1.8 Limitation of the approach

The proposed approach in this thesis has the following limitations that could form the basis for further research:

Limitations pertinent to the recovery scope: the recovery scope of the proposed approach is limited to the structural view of a system as illustrated in Figure 1.1, hence the approach currently can not recover the behavioral and dynamic properties of the software system; ii) the current recovery technique is best suited for the recovery of monolithic software systems, however the technique can be extended to cover the object oriented systems by designing a new domain model to represent the entities and relationships in the object orientation paradigm, and then recover the group of related classes into subsystems; iii) the current approach is not intended for analysis of the distributed systems, since it is meaningless to group in one place a number of related entities that are used in different locations; and iv) the current approach assists on the recovery of module interconnection patterns. The recovery of the architectural styles such as pipe and filter, or client and server requires that the proposed AQL language be augmented to handle typed connector such as files and sockets.

Limitations pertinent to computational complexity: the data mining and graph matching operations that are highlighted in the recovery environment of Figure 1.2(a) are computationally expensive. In Sections 6.5, 7.3.1, 7.4, 8.4.2, and 8.5 of the thesis several techniques have been employed that deal with limiting the complexity and increasing the tractability of the expensive operations in both phases of the recovery environment.

1.9 Thesis overview

The remaining Chapters of this thesis are organized as follows:

Chapter 2: provides an overview of the related work in the area of software architecture representation and recovery.

Chapter 3: discusses the pre-process operations that represent the software system at a higher level of abstraction for the on-line analysis. In these operations the software system is represented as an entity relationship graph. Using a data mining technique two association-based similarity measures at different granularity levels of the system entities are introduced. In this way, the graph of the system is decomposed into graph regions as means to reduce the pattern matching run-time complexity.

Chapter 4: discusses the details of the proposed architecture query language (AQL). An AQL query provides a model of an abstract module-interconnection pattern that denotes a number of structural constrains to be satisfied in the recovered architecture.

Chapter 5: presents means for an AQL query to be presented as a pattern-graph that is to be matched with an entity relation graph of the software system.

Chapter 6: presents a new model for the software architecture recovery based on approximate graph pattern matching. For such a graph matching model, the input graphs consist of a software system graph (defined in Chapter 3) and a pattern-graph (defined in Chapters 4 and 5). Moreover, the characteristics of a multi-phase graph matching algorithm and the corresponding cost functions for graph edit operations are discussed and an example of a two-phase graph pattern matching process is provided.

Chapter 7: presents an overview of the pattern matching algorithms and the corresponding complexity analysis.

Chapter 8: provides the experimentation with a number of middle-size industrial software systems. The experimentations are divided into off-line analysis and on-line analysis, as well as evaluating the stability, quality, and accuracy of the proposed pattern matching technique.

Chapter 9: provides a conclusion for the whole thesis and the possible extensions to the proposed environment.

Appendices: A) Formal definition of an architectural level domain model presented in Chapter 3. B) Formal definitions of graphs presented in Chapter 5. C) Pseudocode for the pattern matching algorithms. D) AQL query for a system that is experimented in Chapter 8. E) Glossary of terms used in this thesis.

Chapter 2

Related work

In this Chapter we present the work related to this thesis in the area of software architecture recovery.

2.1 Architecture recovery

A brief taxonomy of the current architecture recovery frameworks has been presented in [74] as follows:

- *Clustering frameworks*: a parser extracts a relational source model from source code and stores it in a database. Using a number of clustering operations based on properties such as low-coupling and high-cohesion, system components are identified. Rigi [104], PBS [38], and concept lattice techniques [98, 68, 109] are examples in this category.

- *Compliance checking frameworks*: the extraction phase is identical to the previous method. However, in the analysis phase, the analyst first defines his/her assumed high level model of the software in an appropriate form (e.g., modules and interconnection, inheritance hierarchy, design pattern, or architectural style). The tool then checks the degree of conformance between the proposed model and the source model. Software reflexion model is an example [79].

- *Analyzer generators frameworks*: a parser generates an abstract syntax tree and stores it in a repository. A query language is used to generate queries to analyze the source model of the software system for the existence of specific properties. Abstract syntax trees are language dependent, hence, the engineer must be familiar with the language syntax. Genoa [33] is an example.

- *Program understanding frameworks*: this method uses the knowledge of an expert to automatically generate high level views of the system's functionality. The knowledge of the expert is captured in a knowledge-base, and the source model is represented in an abstract syntax tree. A recognition engine then searches through the source model and knowledge-base with the goal of finding possible matches. The result is a hierarchy of recognized patterns which are the user-guided views of the system. DECODE is a tool in this category [83]. Also in [114] a technique that models program understanding as constraint satisfaction problem is presented.

- *Pattern matching frameworks*: a query language is used to model the high-level view of the system either as architectural styles [48], graph of architectural elements [55], or a series of SQL queries [56]. The pattern matching process searches the source model which is either an abstract syntax tree, a repository of architectural elements, or a relational database, respectively, to find the exact or approximate match of the queries in the source model.

In this context, the software architecture recovery technique proposed in this thesis can be categorized as a “pattern matching framework”.

2.2 Architectural views

The significance of software architecture *views* as a means for separating the designer’s concerns has been investigated in the research literature [117, 61, 100, 80, 105, 66].

In a broad sense, a *view* can be defined as *“a projection of a process according to a well-defined characteristic”* [16]. Examples of such characteristics in a software development environment include: roles (e.g., manager, designer), products (e.g., specification, design document), and activities (e.g., review, implementation). In other words, views are the result of applying *separation of concerns* on a development process in order to classify the related knowledge about that process into more understandable and manageable forms.

The selection of an appropriate set of views is a common concern both in software development and in reverse engineering. The different sets of views proposed for specifying or developing a system consists of: data, function, and network [117]; function, process, development, and physical [61]; and module-interconnection, conceptual, execution, and code [100]. The functional view seems to be the central view. The data view is important in the design of a database for an information system. The process view is required in the design of concurrent systems. The development view is mostly required for task assignment and scheduling in project management. The network view (physical view) is essential in designing distributed systems. The user-interface view is helpful in designing interactive systems.

During the reverse engineering, it is ideal to recover the same set of views of a system that are also appropriate for its development. Unfortunately, reverse engineering is a much more difficult task than forward engineering. Recovering some aspects, e.g., functionality, of a large and poorly documented legacy system is a non-trivial task (if not impossible). The structural view is considered to be the most appropriate architectural view to be recovered [38, 1, 116]. Other views to be considered are data and behavior [28].

In this respect, the scope of the architectural recovery approach in this thesis has been restricted to structural view of the system and the features of the system to be recovered have been described in Section 1.5 and are also illustrated in Figure 1.1.

2.3 Architecture description languages

An Architecture Description Language (ADL) is used to document architectural information in order to describe and analyze a system. From the linguistic point of view, an ADL is characterized by six properties [97]:

i) *composition/decomposition*: integration of system components into larger subsystems, decomposition of a system into its constituents, and integration of architectural styles; ii) *abstraction*: defining abstract views of high-level or low-level design; iii) *reusability*: using generic patterns of components and connectors; iv) *configurability*: changing a software's structure independent of the components; v) *heterogeneity*: integrating different styles in one system, or integrating modules written in different languages; and vi) *analysis*: reasoning about the system by checking architectural properties, providing metrics, or simulating run-time characteristics.

A variety of architectural features have been proposed or implemented on ADLs such as: Unicon [96], Rapide [70, 71], ACME [43], ARDEC DSSA [103], Wright [14], ControlH [20], Polylith [82], DECODE [83], Kaptur [17], and also in [67]. However, no single ADL can handle a good portion of these features.

Finally, Module Interconnection Languages (MIL) [81] provide syntactic and semantic means for integrating separately developed modules in a distributed environment. A variety of MILs exist which differ in aspects such as: type of interfacing, complex data-type handling, and language independence. Polylith [82] is an example of an MIL.

In this context, the design of the proposed architecture query language (AQL) has been influenced by the structural design of ADLs and is used to specify the abstract architectural components and connectors which are then instantiated by the actual subsystems (or modules) and their interconnections during the pattern matching process.

2.4 Data mining

Data mining or Knowledge Discovery in Databases (KDD), refers to a collection of algorithms for discovering or verifying interesting and non-trivial relations among data in large databases [37]. A substantial number of data mining approaches in the related literature are based on extensions of the *Apriori algorithm* by Agrawal [12]. These approaches are pertinent to the concept of *market baskets* (or *transactions*[1]) and their contained *items*. A market basket (or simply basket) contains different kinds of items, where the quantity of items of the same kind in the basket is not considered.

The data mining algorithms search the data in large databases to extract frequently occurring patterns, trends, and generalizations about the data items without user intervention. In this context, the interesting relationships may be discovered among groups of items in baskets (*association rules*) [12], among sequences of groups of items in baskets (*sequential patterns*) [13], or among the time of occurrences of transactions (*time-series clustering*) [11].

The *association rules* express the frequency of pattern occurrences such as 30% of baskets that contain the set of items X also contain the set of items Y (shown as $X \Rightarrow Y$). The Apriori algorithm can be used to discover the association rules in two steps. The first step extracts all combinations of items where the number of common container baskets for each combination exceeds a minimum level (each combination is known as a *frequent itemset*). The second step generates association rules using

[1]The notion of a transaction in the data mining context emphasizes on the containment properties, which is different from the notion of a transaction in distributed systems domain which emphasizes on the communication properties.

such frequent itemsets. The general idea is that if, for example, {A,B,C,D} and {A,B} are frequent itemsets, then one can determine whether the rule $\{A, B\} \Rightarrow \{C, D\}$ holds by computing the ratio $r = \frac{\text{no. of common baskets}(\{A,B,C,D\})}{\text{no. of common baskets}(\{A,B\})}$ known as *confidence* r. The rule holds only if $r >$ *minimum confidence*.

The reverse engineering approaches using data mining are very few. Montes and Carver [31] use data mining association rules and a visual representation model to graphically present the subsystems that are identified from the database representation of the subject system. In contrast, the approach taken in this thesis uses a by-product of association rules, by considering frequent-itemsets along with their container baskets. This information is used to encode the structural property of the groups of entities with maximum-level of interaction as a similarity measure between system entities.

2.5 Concept lattice analysis

The *mathematical concept analysis* was first introduced by Birkhoff in 1940 [21]. In this formalism, a binary relation between a set of "objects" and a set of "attribute-values" is represented as a lattice. Recently, the application of concept analysis in reverse engineering has been investigated [98, 68, 109]. In such applications, a *formal concept* is a maximal collection of objects (i.e., system functions) sharing maximal common attribute-values (i.e., called/used functions, datatypes, variables). A *concept lattice* can be composed to provide significant insight into the structure of a relation between objects and attribute-values such that each node of the lattice represents a concept. However, even in medium software systems (+50 KLOC) the concept lattice becomes so complex that the visual characteristic of the lattice is obscured. In such cases, the researchers seek automatic partitioning algorithms to assist the user in finding distinct clusters of highly related concepts.

The steps of using concept lattice for the modularization of a software system have been presented in [98] as follows. First, a matrix of functions and their attribute-values is built. Second, based on this matrix a *concept lattice* is constructed, using a bottom-up iterative process. Finally, a collection of the *concept partitions* is identified, where each partition is a group of disjoint sets of concepts, and the attribute-values in each set of concepts have large overlap. Each partition corresponds to a potential decomposition of the system into modules.

Sif [98] uses a repair technique by adding extra relations to make the generated concept lattice *well-formed* in order to provide easier partitioning. The main drawback of this approach is the large number of generated partitions that requires high user-involvement for reducing the number of partitions to a manageable set for investigation. Snelting [68] uses a technique called "horizontal decomposition" to partition a lattice of procedures and variables into modules. However, the overwhelming number of interferences between concepts in the lattice of a real system prevents such an horizontal partitioning.

In contrast to concept lattice approaches, this thesis defines a similarity measure which encodes the structural characteristics of the neighboring concepts and use this metric to collect the groups of closely related concepts into one module or subsystem.

2.6 Clustering

The pattern-based software architecture recovery presented in this thesis can be also viewed as a clustering technique provided that in the AQL query specification of the architecture the user does not constrain the number of interactions between the modules or subsystems.

The cluster analysis is defined as the process of classifying entities into subsets that have meaning in the context of a particular problem [54]. The clustering techniques are designed to extract group of related entities. However, the choice of a technique affects the detected clusters, which may or may not be the existing structure. The clustering approaches usually determine a "similarity metric" and a "clustering algorithm". Wiggerts [112], Anquetil [15], and Tzerpos [107] have surveyed different aspects of clustering algorithms for software systems.

A *similarity* measure is defined so that two entities that are alike possess higher similarity value than two entities that are not alike. There is a number of similarity measures proposed in the literature, and Wiggerts provides a summary of different categories namely *association coefficients, correlation coefficients*, and *probabilistic measures* [112]. Based on the size ratio of different unions and weights of the sets of shared features, a variety of association based similarity metrics have been suggested [36] such as *Jaccard* and *matching coefficient*[2].

This thesis proposes a new association-based similarity metric in Chapter 3 which has two advantages over the existing association-based similarity metrics:

[2] $Jaccard = \frac{|A \cap B|}{|A \cup B|}$ and $Maching = \frac{|(A \cap B) \cup C|}{|D|}$, where A and B are the sets of features for two entities, C is the set of features not in either entities, and D is the set of whole features.

i) it identifies the members of a group of maximally related entities in a system; and ii) it considers the datatypes and variables as members of a group including functions, as opposed to considering them as attribute-values of functions which cause only the functions to be grouped.

Lakhotia [63] provides a unified framework that categorizes and compares the different software clustering techniques. Furthermore, the clustering-based approaches to software architecture recovery can fall into two groups. The first group of approaches utilizes automatic or semi-automatic techniques [89, 59, 15, 26, 30, 109, 72, 107, 112, 52, 62] using a similarity metric and a clustering algorithm (e.g., agglomerative, optimization, graph-based, or construction) to partition the system into groups of related entities [112]. The second group of approaches [38, 78] is based on tool usage, domain knowledge, and visualization means, to perform a user/tool cooperative clustering process, and to view and evaluate the properties of the clustered system. Such techniques have been proven useful in handling large systems [38].

In [72] a partitioning method is used to partition a group of system files into a number of clusters. The method uses a hill-climbing search to consider different alternatives based on neighboring partitions, where the initial partition is randomly selected. In comparison, our method carefully finds a collection of rather separated and highly qualified sub-spaces that can be viewed as an initial partition of clusters, and then a search algorithm selects a sub-optimal group of files for each cluster. Therefore, the chance of being trapped in a local optimum caused by random partitioning and hill-climbing search, is eliminated.

In [106], a number of system structural properties are used to cluster the system files into a hierarchy of clusters. The method uses subgraph dominator nodes to find subsystems of almost 20 members, and builds up the hierarchy of subsystems accordingly. To simplify the computation, the interactions of more than 20 links to/from a file are disregarded. In contrast, our technique does not assume any pre-existing structure for the system such as directory structure. Instead, it relies on an overall data/control flow dependencies among the system entities to be used for clustering.

In the PBS approach [38], the user defines a containment structure for a hierarchy of subsystems which is derived from: developers, documentation, directory structure, and naming conventions. The tool then reveals the relations between subsystems and represent the system as layouts in HTML pages for the user's inspection and manipulation. In the Rigi tool [78], the extracted facts in the form of RSF tuples are represented as an entity-relation graph of attributed boxes and arrows. The tool then provides interactive facilities for graph filtering and clustering operations to build and explore subsystem hierarchies. In a semi-automatic method [59], an interactive clustering environment uses a suite of known clustering techniques and the Rigi tool, to incrementally detect components of a system.

In comparison, we propose an interactive software architecture recovery technique and an environment that emphasizes on pre-processing the raw system data to a level that either the tool or the user can perform the recovery operation. Visualization of the graphs whose edges are labeled by quantized association values allows the evaluation and fine-tuning of the automatically generated system architecture.

2.7 Graph matching

The architectural recovery technique proposed in this thesis is based on approximate graph matching, hence this Section outlines the related work in this area. *Graph matching* refers to algorithms for comparing two graphs G_1 and G_2 [95, 25, 108, 75], by means of a function f that maps the nodes and edges of G_1 onto the nodes and edges of G_2 ($f : G_1 \mapsto G_2$). Three types of mappings between two graphs G_1 and G_2 are of particular importance, namely: i) *homomorphism* where f allows two or more nodes in G_1 map to one node in G_2; ii) *monomorphism* (or *subgraph isomorphism*) where f allows one node in G_1 to match only with one node in G_2; and iii) *isomorphism* where f allows one-to-one matching in both directions, i.e., f is one-to-one from G_1 to G_2 and f^{-1} is one-to-one from G_2 to G_1 [95]. Furthermore, graph matching can be classified as *exact* or *approximate.* In *exact graph matching*, the problem is to find a subgraph of graph $G1$ that is isomorphic with another graph $G2$. However, in most real applications due to the effect of noise, distortion, sampling error, or lack of a known or fixed pattern, exact matching is not feasible. In such cases, finding a subgraph of the input-graph that is similar to a given pattern-graph, within the boundary of a threshold value, is the primary objective.

A number of researchers have investigated the application of graph matching in different problem domains. Messmer and Bunke [75] compare an input graph with a collection of prototype graphs by first decomposing the prototypes into primitive graphs which are stored in a database, and then comparing them against the primitives of the input graph. Eshera and Fu [34] decompose the matching graphs into simple trees to be matched. Shapiro and Haralick [95] define a graph structural description with weighted nodes and edges to compute the cost for inexact matching. Bunke and Allermann [25] use graph edit operations and generate a state space to be searched for a minimum path. 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 by Eshera.

2.8 Constraint satisfaction

The software architecture recovery approach presented in this thesis is also related to the approaches that model the problem of program understanding as a *constraint satisfaction problem* (CSP). In CSP the values of a set of variables are restricted by the constraints that are defined between the variables. A solution to a CSP is an assignment of values to variables such that the constraints are satisfied. In the CSP problems the constraints are considered as "hard" that can not be violated.

Woods [114] generalizes the problem of program understanding as an instance of the constraint satisfaction problem. The approach uses templates (as patterns to be recovered) to model the structure of the pieces of source code. A search algorithm instantiates the variables in the templates with the source code entities so that, the structural constraints are satisfied. In this approach two sets of constraints are considered, i.e., one set between source code entities and one set between template variables. In comparison, the AQL query presented in this thesis and the templates in the Wood's approach have been intended for the same objective but with different languages and levels of abstraction. However, in our approach the constraints can be violated with some cost.

In [92] an approach to software architecture recovery is presented that uses an extension to the CSP problem known as *Valued Constraint Satisfaction Problem* framework (*VCSP*) [93], that allows *over-constraint* problems to be dealt. In the VCSP framework a *cost function* assigns a cost for violation of each constraint, and the cost for a certain *value to variable* assignment is the overall cost of constraints that are violated by such an assignment. The goal is to find a complete assignment of minimum cost.

In comparison, the approach in this thesis uses graph matching that assigns a cost to each graph edit operation (i.e., edge/node insertion/deletion) and the goal is to find a match between a graph that represents the software system and a pattern-graph that represents the architecture in query so that the overall graph edit cost is minimized. Moreover, the graph model of a software system is more intuitive than the VCSP model since the data/control flow dependencies can be directly matched and dealt with.

2.9 Architectural pattern matching

The following approaches to software architectural recovery use search techniques to recover a defined pattern in a software system.

Kazman and Carriere [56] propose *Dali* as a workbench that allows different light-weight tools and techniques to integrate for an architectural recovery task. Dali extracts *elements* (function, files, variables, objects), a collection of *relations* (e.g., function calls), and a set of attributes of elements and relations (e.g., function calls function N times), and stores them in a relational database. A pattern consists of a collection of SQL queries that have been integrated via Perl expressions. The primitive SQL queries collect the architectural components and their derived relations by querying the relational database. The recovery process requires the involvement of the user who is familiar with the system's domain (domain expert) and has experience with composing SQL queries. In order to recover the architecture of a system the user composes two sets of pattern queries namely "common application patterns" that are used for all systems and "application-specific patterns" that require knowledge about the domain's reference architecture. In each set of queries the smaller entities are collapsed into larger components, and relations between components are derived. In contrast, the approach in this thesis presents a modular pattern of the software system using the more expressive AQL queries and a tool provides useful information on how to generate the pattern, hence requires less user involvement.

Kazman and Burth [55] introduce an interactive architecture pattern recognition to recover user defined patterns of architectural elements in a system. The system is modeled as a graph of architectural elements, i.e., components and connectors. Each component or connector is defined using common features, namely static and temporal features, causing the elements to be treated in the same way [57]. The user defines an architectural pattern or style as a graph of elements. The tool then searches to identify instances of that graph in the system model. The tool uses the constraint satisfaction paradigm [114] to restrict the search space. The hard/soft features of the elements allow to relax the exact matching (i.e., approximate matching). The approach provides statistics about the regularity of a system in terms of its coverage by a particular pattern. This approach has very interesting capabilities in modeling the architectural elements. The pattern in this approach describes the interaction between individual elements in the system model as opposed to our approach that the AQL query defines a macroscopic pattern on the groups of system entities and the interaction among the groups of entities.

Murphy and Notkin [79] have proposed the software reflexion model to assist the user in testing his/her mental model of the system. The user employs a textual declarative form to define a high-level model of the system, and link this model to the source model. The source model is a call graph or an inheritance hierarchy. A software reflexion model is then computed to determine where the user's high-level model conforms with the source model and where does not conform. The user interprets the reflexion model and defines new relations based upon the results. Regular expressions are used in the forms to facilitate the link of a group of source model entities to a high-level entity. 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.

Harris et al. [48] identify architectural styles (about nine styles) in source code. The method uses an annotated AST of the system as the search domain and an architectural query language, built on top of the Refine language, that codifies the desired architectural styles. A number of style recognition queries (around 60) constitute the base of the recognition process. A specialized query is composed to search for specific style related properties in the source model. This query triggers a set of more specific style queries as subgoals, and then reports on the degree of success in recognizing that style and its code-coverage. In a similar approach, Fuitem et al. [39, 40] use recognizers and flow analysis techniques in architectural recovery. The expected architecture has a hierarchical model with components and connectors at different levels (e.g., system, program, and module), and different views at each level (e.g., system, module, task, code, data/call graphs).

The two approaches above, use program understanding techniques at a higher-level of abstraction to recognize architectural styles using plan-like queries. However, in order to compose a query the engineer needs to be familiar with the system using the recognizers in the repository. In contrast, the abstract query in our approach needs no repository of recognizers and allows the engineer to recover a modular view of the whole software system incrementally.

Dean and Cordy [32] describe a software architecture using a pattern language that is based on typed nodes and connections. The pattern language can model the architectural structure of a system by defining the semantics of individual components and the system as a whole. At a higher level, a framework can be defined to abstract away the details of particular components and provide a means of categorizing architectural paradigms.

Kontogiannis [58] et al. propose a program understanding approach namely concept-to-code pattern matching, where a concept language captures the abstract properties of a desired code fragment. The pattern matching process is based on the *Markov* model and the similarity measure between an abstract pattern and a piece of code is defined in terms of the probability that the abstract pattern can generate that piece of code. Dynamic programming has also been used to reduce the complexity of the required computations. This approach differs from our approach at the level of abstraction used and the employed pattern matching technique.

Chapter 3

System representation

As the first step of any software analysis technique the source-code of the software system is parsed and represented as an Abstract Syntax Tree (AST) or as a set of entity relationship tuples. Since, the source-code representation is too detailed to perform any meaningful architectural analysis, it must be represented at a higher level of abstraction. In this Chapter, a domain model is presented to define an abstract representation of the system entities and the relationships that are suitable for architectural analysis. This domain model provides means for representing a software system as an *Attributed Relational Graph* (ARG) [95, 25, 35]. However, searching and analyzing such a graph representation of a large software system is inherently intractable. In order to address the tractability problem and to provide an incremental analysis of the system architecture, the graph representation of the software system is partitioned into a number of smaller subgraphs using the association property.

This Chapter is organized as follows. First, a software system is modeled as an attributed relational graph using an abstract domain model. Second, the notion of maximal association is presented and two techniques for identifying entities that are related by maximal association are introduced. Third, two association based similarity metrics at file-level and function-level granularity are defined. Finally, a technique for decomposing the software system's graph into a collection of graph regions is presented.

3.1 Graph representation of a software system

Modeling system entities and relationships as an attributed graph has been traditionally used in image processing and pattern matching domains [95, 25, 35]. Modeling a software system as an attributed graph has also been adopted by a number of software analysis approaches [19, 49, 50, 72, 78], where the nodes and edges of the graphs and their attributes are specified by a domain model.

3.1.1 Abstract domain model

A *domain model* provides a schema to represent the software system entities and their interactions using different diagrammatic techniques such as entity relation graphs, module interconnection graphs, structure charts, program dependency graphs, and abstract syntax graphs [73]. For example, the domain model of a programming language can be used to obtain the entity relation graph of a system at the source-code level by considering: i) source-code constructs as instantiations of the domain model classes (i.e., file, function, statement, expression, type-specifier, and variable); and ii) relationships between source-code entities as instantiations of the associations between domain model classes. Such a *source-level* domain model for the C programming language is presented in Appendix A. The instantiation of the "classes and associations" of the domain model into "objects and relationships" is the result of parsing a software system. Recent approaches [49] use XML notation [7] to define a domain model using Domain Type Definition (DTD) that is derived from the grammar of the programming language being modeled.

In reverse engineering, the level of granularity of the selected source-code representation is determined according to the purpose of the analysis. In the context of software architecture recovery, functions, global variables, and aggregate datatypes have been extensively used [63, 30, 15] as the proper granularity for *function-level* system analysis. However for *file-level* analysis, file and directory information have been used instead [24, 72, 106]. The architecture recovery approach in this thesis aims at analyzing a software system at both levels, hence the proposed domain model must cover the entities and relations for both levels of architectural recovery.

In Figure 3.1 the UML class diagram of the proposed domain model, namely the *abstract domain model* is illustrated. In this model the different types of entities are a subset of the types of entities in the software system's source-code, and each relation in the abstract domain model is an aggregation of one or more relations in the software system's source-code. The advantage of this domain model is that it is simpler than the detailed source-code domain model; it is language independent for procedural programming paradigm; and yet it is adequate for architecture level analysis. The class attributes of the different entities in the abstract domain model are presented in Tables 3.1 and 3.2.

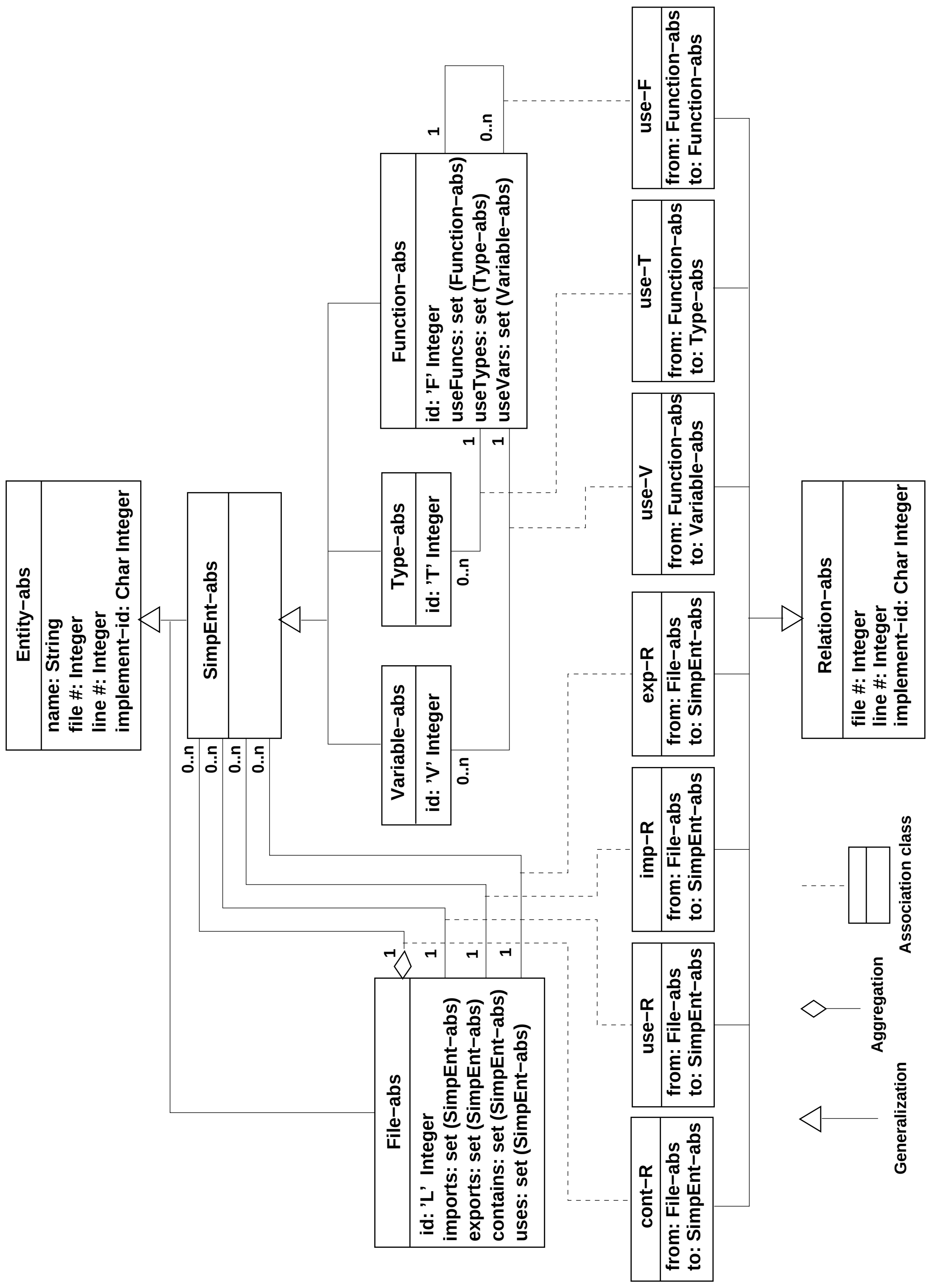


Figure 3.1: The class diagram of an abstract domain model suitable for architectural recovery task. This domain model is an abstraction of the source-level domain model that is presented in Appendix A. This domain model defines the node and edge attributes of the source-graph G^s to be discussed in Section 3.1.2.

Entity-abs		
Attribute	*Example*	*Description*
name	"foo"	Name of the entity in the source-code
file #	5	File number of the source-code file where entity is defined
line #	79	Line number of the entity in the source-code file
implement-id	13	Unique identifier of the object (entity) in the source-level domain model that implements the entity-abstraction in the abstract domain model. The object of an "id" is returned by the function Obj(id)

Table 3.1: Description of the class Entity-abs.

In Figure 3.1, 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. The class *SimpEnt-abs* is used to separate a file entity from other types of entities since they belong to different granularity levels. 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. The relations in the abstract domain model are categorized into function-level and file-level relations as defined below. The formal definitions of these relations are provided in Appendix A.

The entities in the abstract domain model are denoted as *file-abstraction, function-abstraction, datatype-abstraction*, and *variable-abstraction.*

File-abs		
Attribute	*Example*	*Description*
id	L3	Unique identifier of a file-abstraction object that is implemented by a source-file with "id" l3
imports	{F3, T7, ..}	Set of imported entity-abstraction by the relation *imp-R*
exports	{V2, T8, ..}	Set of exported entity-abstraction by the relation *exp-R*
contains	{F9, V1, ..}	Set of contained entity-abstraction by the relation *cont-R*
uses	{F13, T1, ..}	Set of used entity-abstraction by the relation *use-R*
Function-abs		
Attribute	*Example*	*Description*
id	F5	Unique identifier of a function-abstraction that is implemented by a source-code function with "id" f5
useFuncs	{F2, ..}	Set of function-abstraction that are related to this function-abstraction by the relation *use-F*.
useTypes	{T5, ..}	Set of datatype-abstraction that are related to this function-abstraction by the relation *use-T*.
useVars	{V6, ..}	Set of variable-abstraction that are related to this function-abstraction by the relation *use-V*.
Type-abs		
Attribute	*Example*	*Description*
id	T9	Unique identifier of a datatype-abstraction that is implemented by an aggregate-type or array-type.
Variable-abs		
Attribute	*Example*	*Description*
id	V6	Unique identifier of a variable-abstraction that is implemented by a global-variable

Table 3.2: Description of the classes File-abs, Function-abs, Type-abs, and Variable-abs. The relations *imp-R, exp-R*, *cont-R*, and *use-R* will be defined later in this Section.

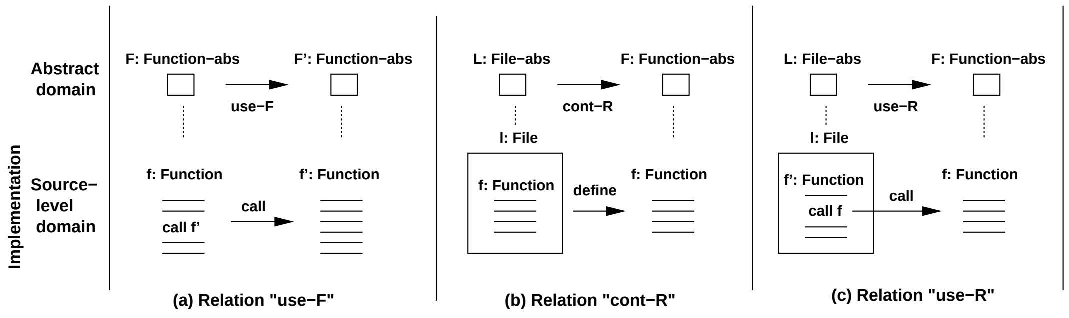


Figure 3.2: The correspondence between the entities and relationships in abstract domain model (Figure 3.1) and source-level domain model of a software system. The source-level domain model is presented in Appendix A.

Function level relations

Relation *use-F* : is defined between two function-abstractions F and F', denoting that the implementation of F (i.e., a source-code function f) calls the implementation of F' (i.e., source-code function f'). Figure 3.2(a) illustrates such a relationship.

Relation *use-T* : is defined between a function-abstraction F and a datatype-abstraction T, denoting that the implementation of F (i.e., function f) updates/reads the value of a variable v, and the variable v is of type aggregate-type/array-type t, and t is the implementation of the datatype-abstraction T.

Relation *use-V* : is defined between a function-abstraction F and a variable-abstraction V, denoting that the implementation of F (i.e., function f) updates/reads the value of the implementation of V (i.e., global variable v).

File level relations

Relation *cont-R* (interpreted as *contain-resource*): is defined between a file-abstraction L and either a function-abstraction F, or a datatype-abstraction T, or a variable-abstraction V, denoting that:

i) the implementation of L (i.e., source-file l) defines the implementation of F (i.e., function f). This relation is illustrated in Figure 3.2(b);

ii) the implementation of L (i.e., source-file l) either defines the implementation of T (i.e., the aggregate-type/array-type t), or a library-file h globally defines the datatype t but the source-file l "has the highest number of references" (i.e., uses) to datatype t among all other source-files; or

iii) the implementation of L (i.e., source-file l) either defines the implementation of V (i.e., the global-variable v), or a library-file h globally defines variable v but the source-file l "has the highest number of references" (i.e., uses) to variable v among all other source-files.

In this context, a file-abstraction is called a *composite entity* and a function-abstraction, a type-abstraction, or a variable-abstraction is called a *simple-entity* such that a composite entity contains a set of simple entities.

Relation *use-R* (interpreted as *use-resource*): is defined between a file-abstraction L and either a function-abstraction F, or a datatype-abstraction T, or a variable-abstraction V, denoting that:

i) the implementation of L (i.e., source-file l) defines a function f', and function f' calls function f, and function f is the implementation of function-abstraction F. This relation is illustrated in Figure 3.2(c);

ii) the implementation of L (i.e., source-file l) defines a function f', and function f' updates/reads the value of a variable v' whose type is the aggregate-type/array-type t, and t is the implementation of datatype-abstraction T; or iii) the implementation of L (i.e., source-file l) defines a function f', and function f' updates/reads the value of global variable v, and v is the implementation of variable-abstraction V.

Relation *imp-R* (interpreted as *import-resource*): is defined between a file-abstraction L and an entity-abstraction R (i.e., function-abstraction / datatype-abstraction / variable-abstraction), denoting that L uses R but does not contain R[1].

Relation *exp-R* (interpreted as *export-resource*): is defined between a file-abstraction L and an entity-abstraction R (i.e., function-abstraction / datatype-abstraction / variable-abstraction), denoting that L contains R and another file-abstraction L' uses R.

3.1.2 Source graph

In this Section, the graph representation of a software system at architectural analysis level, namely *source-graph* is discussed. This is one of the core data models for the proposed graph pattern based software architecture recovery. In this thesis, the notation for *Attributed Relational Graph* (ARG) that is presented in [34] is adopted to define all graphs. The attributed relational graphs are frequently used

[1]In the graph representation of the entity-abstraction and relation-abstraction the relation $(L, R) \in$ *use-R* is shown as an edge: $L \xrightarrow{use\text{-}R} R$. However, when the same relation is represented as the import relation $(L, R) \in$ *imp-R* then the corresponding edge is shown as: $L \xleftarrow{imp\text{-}R} R$. Similar notation applies for the *exp-R* relation which is defined next.

for modeling systems in graph matching problems [75, 34, 95, 25].

the attributed relational graph representation of the source-graph is a six-tuple $G^s = (N^s, R^s, A^s, E^s, \mu^s, \epsilon^s)$[2] that is defined as:

- N^s : $\{n_1, n_2, ..., n_n\}$ is the set of attributed nodes, obtained from the abstract domain model.
- R^s : $\{r_1, r_2, ..., r_m\}$ is the set of attributed edges, obtained from the abstract domain model.
- A^s : alphabet for node attributes and node attribute values such as node labels, node types, and their values.
- E^s : alphabet for edge attributes and edge attribute values such as edge labels, edge types, and their values.
- $\mu^s : N^s \rightarrow (A^s \times A^s)^p$: a function for returning the "node attribute, node attribute value" pairs where p is a constant and denotes the number of node attributes.
- $\epsilon^s : R^s \rightarrow (E^s \times E^s)^q$: a function for returning "edge attribute, edge attribute value" pairs where q is a constant and denotes the number of edge attributes.

In the source-graph G^s the node and edge attributes are obtained from the abstract domain model presented in Section 3.1.1, where each graph node is an

[2]Without loss of generality, we can refer to source-graph $G^s = (N^s, R^s, A^s, E^s, \mu^s, \epsilon^s)$ as $G^s = (N^s, R^s)$.

object of a subclass of the *Entity-abs* class, and each graph edge is an object of a subclass of the *Relation-abs* class. A node n_j represents either a file-abstraction, function-abstraction, datatype-abstraction, or variable-abstraction[3]. An edge r_y represents a relation-abstraction such as *use-F, use-T*, or *use-V* for function-level analysis; and *cont-R, use-R, imp-R*, or *exp-R* for file-level analysis. In the source-graph G^s the major attributes for the nodes and edges are:

- *label:* denotes: i) a full path-name as a unique name for each entity in the software system; ii) a unique object identifier to refer to an entity, e.g., F4, L6, T32; and iii) source and sink nodes to identify an edge, e.g., (n_2, n_8).
- *type:* denotes the type of each node or edge in the graph. For example, *File-abs, Function-abs, Type-abs, Variable-abs* are different types for nodes; and *use-F, use-T, use-V, cont-R, use-R, imp-R, exp-R* are different types for edges.
- *location:* denotes the *source file number* and *line number in file* where the entity (e.g., function F4) or the relation between two entities (e.g., use-V) is implemented in the software system.

Graph edges correspond to a ternary relation *node-type*× *edge-type*× *node-type* and are represented as the triples such as (*Function-abs, use-F, Function-abs*) and (*Function-abs, use-V, Variable-abs*) with the intuitive interpretations of: *function calls function* and *function updates/reads variable*, respectively.

[3]For simplicity, in the rest of this Chapter, *file-abstraction, function-abstraction, datatype-abstraction*, and *variable-abstraction* are referred to as *file, function, datatype*, and *variable*, respectively.

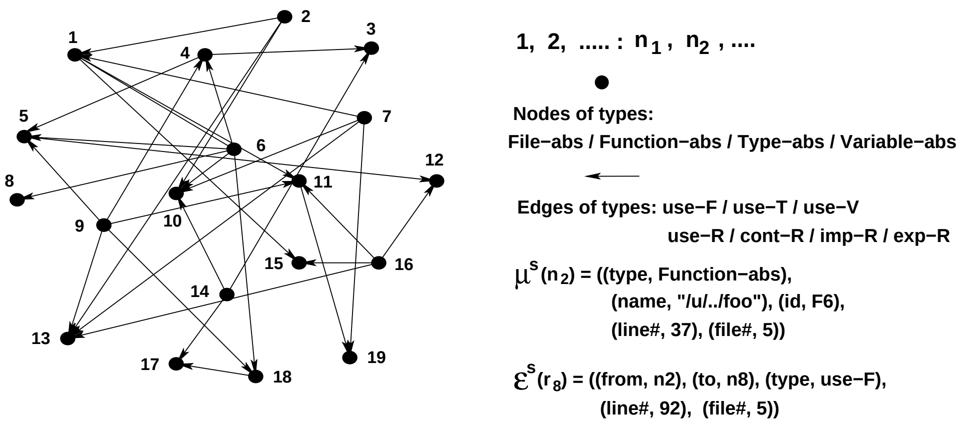

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
1, 2, ... : n1 , n2 , ...
Nodes of types:
File−abs / Function−abs / Type−abs / Variable−abs
Edges of types: use−F / use−T / use−V
use−R / cont−R / imp−R / exp−R
μs(n2) = ((type, Function−abs),
(name, "/u/../foo"), (id, F6),
(line#, 37), (file#, 5))
εs(r8) = ((from, n2), (to, n8), (type, use−F),
(line#, 92), (file#, 5))

Figure 3.3: An attributed relational graph representation of a source-graph $G^s = (N^s, R^s, A^s, E^s, \mu^s, \epsilon^s)$ according to the abstract domain model Figure 3.1.

Figure 3.3 illustrates the ARG of a small source-graph with 19 nodes and the examples of node and edge labeling functions μ^s and ϵ^s that are explained below:

- $\mu^s(n_2)$ = ((type, *Function-abs*), (name, "/u/../foo"), (id, F6), (line#, 37), (file#, 5)) indicating that node 2 of the source-graph G^s is an entity of type *Function-abs* with name *"/u/../foo"* and id F6 and it has been defined in line 37 of the source file 5; and

- $\epsilon^s(r_8) = ((from, n_2), (to, n_8), (type, use\text{-}F), (line\#, 92), (file\#, 5))$ indicating that the edge r_8 of the source-graph G^s is an object of type *use-F* that relates the function n_2 (discussed above) to the function n_8 with a function-call relation in line 92 of file 5.

3.2 Computing maximal association

The proposed approach to software architecture recovery aims at extracting a collection of components (as modules or subsystems) with maximal intra-relation association according to data/control flow dependencies between the component elements. *Maximal association* can be extracted by data mining and concept lattice analysis and is considered as an interesting property for grouping the entities into cohesive modules [98, 68, 109]. This property is mostly used to visualize the structure of relations among groups of entities in small programs using a lattice, that provides insight into the program under analysis. In this thesis we use the notion of maximal association to define two similarity metrics. Informally, maximal association is defined in a group of entities in the form of a maximal set of entities that all share the same relation to every member of another maximal set of entities.

For every set of functions, denoted as $\mathcal{F}$, we can determine a set of shared entities , denoted as $\mathcal{E}$, where every function f in $\mathcal{F}$ has a relation *rel* to an entity e in $\mathcal{E}$. For example, two functions f and g can share the datatype t and variable v by the relations *use-T* and *use-V*, respectively. The operation *sh-ents*($\mathcal{F}$) returns the set of shared entities $\mathcal{E}$ for the set $\mathcal{F}$ as follows:

$$\textit{sh-ents}(\mathcal{F}) = \{e \mid \forall f \in \mathcal{F};\ \exists rel : X \bullet X \in \{\textit{use-F},\ \textit{use-T},\ \textit{use-V}\} \wedge (f,\ e) \in rel\}$$

Similarly, for every set $\mathcal{E}$ of entities we can determine a set of functions $\mathcal{F}$, where every function f in $\mathcal{F}$ has a relation *rel* to an entity e in $\mathcal{E}$. The operation *sh-*

funcs($\mathcal{E}$) returns the set of sharing functions $\mathcal{F}$ for the set $\mathcal{E}$ as follows:

$$\textit{sh-funcs}(\mathcal{E}) = \{f \mid \forall e \in \mathcal{E};\ \exists rel : X \bullet X \in \{\textit{use-F},\ \textit{use-T},\ \textit{use-V}\} \wedge (f,\ e) \in rel\}$$

A set of functions $\mathcal{F}$ and a set of entities $\mathcal{E}$ are related by maximal association, iff:

$$\mathcal{F} = \textit{sh-funcs}(\mathcal{E}) \quad \wedge \quad \mathcal{E} = \textit{sh-ents}(\mathcal{F})$$

In this form, no larger set of functions $\mathcal{F}'$ ($\mathcal{F}' \supset \mathcal{F}$) exists such that $\mathcal{F}'$ and the set of entities $\mathcal{E}$ are related by maximal association. Similarly, no larger set of entities $\mathcal{E}'$ ($\mathcal{E}' \supset \mathcal{E}$) exists such that $\mathcal{F}$ and $\mathcal{E}'$ are related by maximal association.

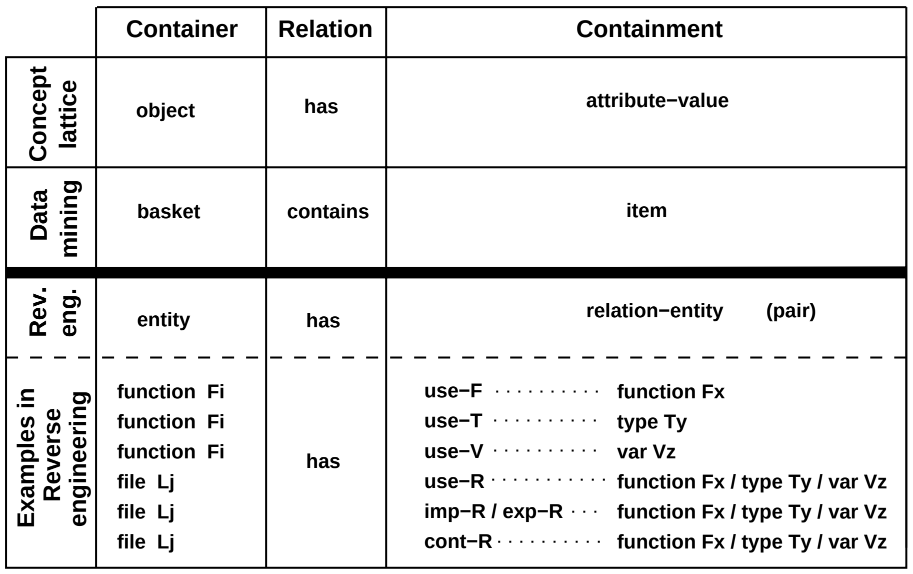
In the following Sections, a technique from mathematical concept analysis and a technique from data mining that extract groups of entities with maximal association are discussed and briefly compared.

3.2.1 Mathematical concept analysis

In mathematical concept analysis a *formal context* is a triple $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$, where $\mathcal{O}$ and $\mathcal{A}$ are the sets of objects and attribute-values, and $\mathcal{R}$ is a binary relation "*has*" between objects and attribute-values. A formal context can be represented as a table of objects and attribute-values, referred to as the *context table.*

A *formal concept* is a *maximal collection of objects sharing maximal common attribute-values.* A formal concept c is represented as a two-tuple $c = (\mathcal{O}, \mathcal{A})$, where $\mathcal{O}$ and $\mathcal{A}$ are called the *extend* and *intend* of concept c, respectively [21].

In order to apply concept lattice analysis on the reverse engineering domain, a mapping from objects and attribute-values in the concept lattice onto the entities



	Container	Relation	Containment
Concept lattice	object	has	attribute−value
Data mining	basket	contains	item
Rev. eng.	entity	has	relation−entity (pair)
Examples in Reverse engineering	function Fi		use−F ·········· function Fx
	function Fi		use−T ·········· type Ty
	function Fi		use−V ·········· var Vz
	file Lj	has	use−R ··········· function Fx / type Ty / var Vz
	file Lj		imp−R / exp−R ··· function Fx / type Ty / var Vz
	file Lj		cont−R ·········· function Fx / type Ty / var Vz

Figure 3.4: Mapping the "object and attribute-value" in concept lattice analysis and "basket and item" in data mining domain onto the "entity and relation-entity" in the reverse engineering domain.

and relations in a software system's domain model must be provided. Figure 3.4 demonstrates such a mapping, where an "object" is mapped onto a system "entity" such as a function or a file, and an "attribute-value" is mapped onto a pair of "relation and entity". For example, a function F1 (as an object) can have a "*use-F* function F2", a "*use-T* type T3", and a "*use-V* variable V2" as the attribute-values of the function F1[4].

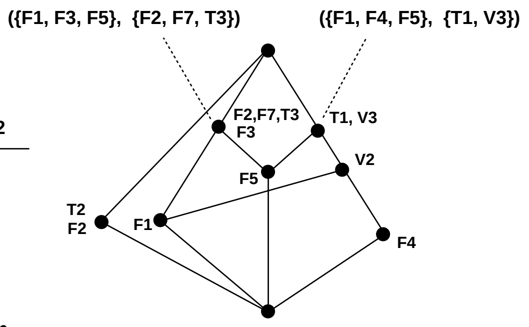
In this mapping, the set of functions $\mathcal{F}$ (as objects $\mathcal{O}$) and the set of "relation and entity" pairs $\mathcal{A}$ (as attribute-values) correspond to a "maximal rectangle" in the context table. Therefore a concept c provides maximal association between the

[4]In this example, the attribute-values function F2, type T3, and variable V2 are stored in the attributes *useFuncs, useTypes*, and *useVars* of the function F1 (an object of class *Function-abs*) in the abstract domain model of Figure 3.1.

Attr−Val / Obj	F2	F7	T3	T1	V2	V3	T2
F1	X	X	X	X	X	X	
F2							X
F3	X	X	X				
F4				X	X	X	
F5	X	X	X	X		X	

Attribute−value	Interpretation
F2	"use−F function F2"
T3	"use−T type T3"
V2	"use−V variable V2"

(a) Context table



(b) Concept lattice

Figure 3.5: An example of a formal context and its corresponding concept lattice.

set of functions in $\mathcal{F}$ and the set of entities in $\mathcal{A}$ by the equations:

$$\mathcal{F} = sh\text{-}funcs(\mathcal{A}) \quad \wedge \quad \mathcal{A} = sh\text{-}ents(\mathcal{F})$$

To simplify the representation of objects and attribute-values we adopt the following: i) an object is represented by its identifier, e.g., function F2 and type T5 are shown as F2 and T5; and ii) an attribute-value consisting of a "relation and entity" pair is replaced by only the entity's identifier, e.g., "*use-V* variable V2" is replaced by V2. Therefore, the term "function F1 $\underline{has}$ *use-V* variable V2" is represented by "F1 *has* V2".

Figure 3.5(a) illustrates the context table of a formal context with five objects F1 to F5, and seven attribute-values F2, F7, T1, T2, T3, V2, V3. In this context table, a concept corresponds to a *maximal* rectangle consisting of particular rows and columns, where every two rows (or two columns) can be swapped without change of concepts. Figure 3.5(b) illustrates the concept lattice of the context table in Figure 3.5(a). A concept lattice can be composed to illustrate the structure of the relations between objects and attribute-values such that each node of the lattice represents a concept. A concept lattice has the following characteristics:

- Each lattice node (i.e., a concept) is labeled with objects (functions) and attribute-values, except the top and bottom nodes that may be unlabeled.
- Every object has all attribute-values that are above it in the lattice (directly above or separated by some links).
- Every attribute-value exists in all objects that are below it in the lattice (directly below or separated by some links).

For example, the node labeled $|_{F3}^{F2,F7,T3}$ in the lattice of Figure 3.5(b) corresponds to the concept $c_x = (\{F1, F3, F5\}, \{F2, F7, T3\})$ which means each of the functions $F1, F3, F5$ shares all attribute-values $F2, F7, T3$. In other words, all the functions $F1, F3, F5$ use the functions $F2$ and $F7$ and use the aggregate datatype $T3$. Also the node labeled "$T1, V3$" in the lattice corresponds to the concept $c_y = (\{F1, F4, F5\}, \{T1, V3\})$ with similar interpretation. Such interesting properties are not easily observable in the context table of a large software system.

In the next Section, we discuss data mining as an alternative technique to extract maximal association and compare it against the concept lattice analysis.

3.2.2 Data mining

The data mining domain was introduced in Section 2.4. The *association rules* in the data mining domain express the frequency of pattern occurrences such as

30% of baskets that contain the set of items X also contain the set of items Y. The association rules are generated by frequent-itemsets and the frequent itemsets can be generated by the *Apriori* algorithm [12]. A frequent itemset has the same interpretation as a concept in the concept lattice analysis. A *k-itemset* is a set of items with cardinality $k > 0$ and a *frequent k-itemset* is a k-itemset (or simply itemset) whose elements are contained in every basket of a group of baskets (called *supporting transactions*). The cardinality of this group of baskets must be greater than a user-defined threshold called *minimum-support*.

In order to apply the data mining techniques on the reverse engineering domain, a mapping from the notions of baskets and items in the data mining domain onto the notions of entities and relations in a software system domain must be provided. The mapping is similar to that of concept lattice analysis discussed earlier. Figure 3.4 illustrates such a mapping, where a "basket" is mapped onto a system "entity" such as function F1 or file L2, and an "item" is mapped onto a pair of "relation and entity". In this mapping the "contains" relation in data mining is mapped onto the "has" relation. For example, a function F2 (as a basket) can have a "*use-F* function F5", a "*use-T* type T3", and a "*use-V* variable V2" as the items of the basket F2.[5]

In order to apply the Apriori algorithm on the source-graph G^s, we define $B(G^s)$ as the *basket representation* of the source-graph $G^s = (N^s, R^s)$:

$B(G^s) = \{b : \textit{Function-abs};\ I : set(\textit{Entity-abs})\ \ |$

[5]Similar to the discussion on concept lattice analysis, in data mining for simplicity of the representation of an item such as "*use-F* function F5" only the identifier "F5" is shown. Therefore, "function F2 <u>*has*</u> *use-F* function F5" is represented as "F2 *contains* F5" and "function F2 <u>*has*</u> *use-V* variable V3" is represented as "F2 *contains* V3".

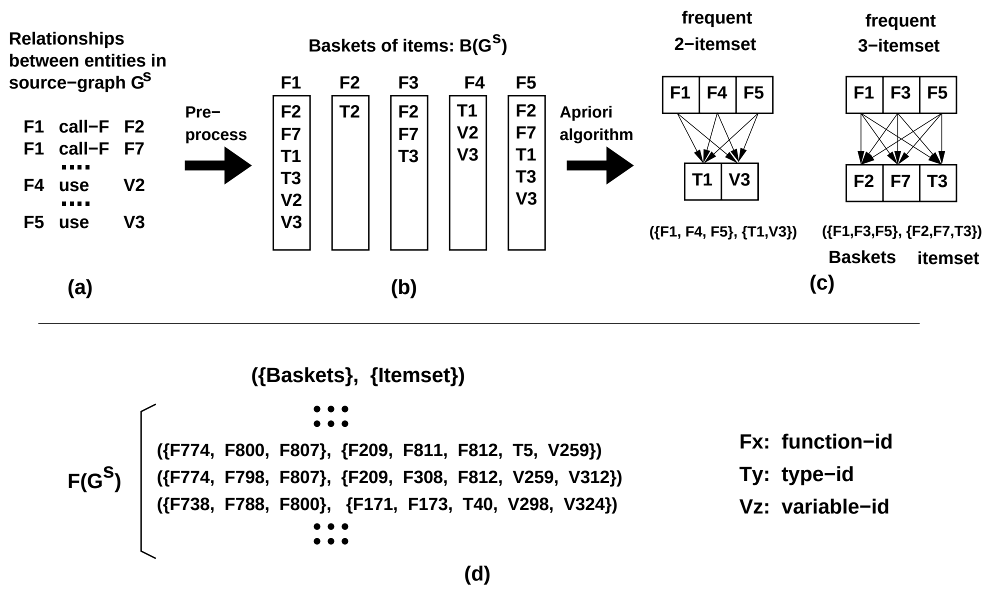


Figure 3.6: (a),(b),(c) Application of data mining in extracting frequent itemsets. (d) Representation of the frequent itemsets for system analysis.

$$b \in N^s \quad \wedge \quad I = b.useFuncs \;\cup\; b.useTypes \;\cup\; b.useVars\}$$

Figure 3.6 illustrates the process of generating frequent itemsets from the source-graph G^s. In Figures 3.6(a) and 3.6(b), the entities and relationships in source-graph G^s are represented as a database of baskets and items $B(G^s)$. In Figure 3.6(c), two frequent-itemsets generated by the Apriori algorithm on $B(G^s)$ are shown. Each frequent-itemset is presented as a tuple ($\{baskets\}$, $\{items\}$), where $\{baskets\}$ is the set of functions and $\{items\}$ is the set of "relation and entity" pairs, such that:

$$\{baskets\} = sh\text{-}funs(\{items\}) \qquad \wedge \qquad \{items\} = sh\text{-}ents(\{baskets\})$$

Hence the set of functions in $\{baskets\}$ and the set of entities in $\{items\}$ are related by maximal association.

In Figure 3.6(c) the resulting frequent itemsets, i.e., $(\{F1, F4, F5\}, \{T1, V3\})$ and $(\{F1, F3, F5\}, \{F2, F7, T3\})$ are the same as the two formal concepts illustrated in Figure 3.5.

The generated frequent itemsets are categorized into groups, based on the cardinality i of the itemset ($i \in [1..k]$) and are stored in a database, denoted as $F(G^s)$, for further analysis in Section 3.3.

Finally, Figure 3.6(d) represents a small portion of frequent 5-itemsets (5 items in each itemset) extracted from a software system's source-graph. The first line is interpreted as: all the functions F774, F800, F807 use functions F209, F811, F812, and use aggregate type T5 and global variable V259. Each *frequent 5-itemset* is equivalent to a *concept* with *intend* size 5.

Data mining versus concept lattice analysis

Both concept lattice and data mining techniques identify groups of entities related by maximal association in the form of "concepts" or "frequent-itemsets", respectively. Below, a discussion on the applicability of these techniques for software modularization is provided.

- The approaches to concept lattice analysis mostly rely on visualizing and understanding the structural properties of the neighboring concepts in the lattice to generate cohesive modules, hence these approaches are highly user-dependent (see the related work in Section 2.5). In this Chapter, we propose a new similarity measure that maps the structural properties of neighboring groups with maximal association generated by data mining techniques to be

Concept Analysis	Object	Attr-value	Concept	Extend	Intend
Data Mining	Basket (Transaction)	Item	Frequent itemset	Support	Itemset

Figure 3.7: The relation between the domains of *concept lattice analysis* and *data mining.*

used for the proposed pattern matching process.

- In concept lattice analysis, the large number of generated concepts is a bottleneck for partitioning algorithms, and there is no implemented mechanism to restrict the number of generated concepts. In data mining techniques the quantity of the generated frequent-itemsets can be controlled by a user-defined parameter minimum-support (i.e., minimum-baskets). For example, in Figure 3.5 the concept $c = (\{F2\}, \{T2\})$ is not generated in the data mining technique, if the value of minimum-support is 2 or more.

Figure 3.7 illustrates the relation between the domains of concept lattice analysis and data mining. In the following Section, we use the maximal association property among the group of system entities to define a new similarity measure between two entities.

3.3 Similarity measure between two entities

In the previous Sections the significance of maximal association property in software modularization using concept lattice analysis to produce cohesive modules was discussed. However, even in medium software systems (+50 KLOC) the concept lattice becomes so complex that the visual characteristic of the lattice is obscured. Therefore, an objective measure that allows to use the maximal association relation in the recovery of cohesive modules in large systems would be necessary.

A *similarity measure* is defined so that two entities that are alike possess higher similarity value than two entities that are not alike. The clustering research literature provides a rich collection of techniques for extracting groups of related software entities using different similarity metrics namely *association metrics, correlation metrics*, and *probabilistic metrics* [36, 54, 112]. The *Jaccard* metric[6] is commonly used as an association similarity metric, and according to an evaluation of a number of similarity metrics in [112], the Jaccard metric produces better clusters than the others, and moreover it is considered more intuitive as a metric.

In this Section, we define the *entity association* similarity measure between two system entities such as functions, datatypes, and variables in the system graph.

Entity association similarity measure

The entity association measure is an extension to the notion of association in the clustering and data mining domains that are briefly compared below:

- *clustering*: the association similarity is defined between two entities as the proportion of the numbers of shared and total attribute-values, Figure 3.8(a).
- *data mining*: the association rule (confidence) is defined between two sets of items as the proportion of the numbers of the shared and total baskets, Figure 3.8(b).

[6] $Jaccard = \frac{|A \cap B|}{|A \cup B|}$, where A and B are the sets of attribute-values for two entities.

Therefore, the association property is defined between either: sharing entities (clustering), or shared entities (data mining). In this context, we are interested in defining an association based similarity measure to apply it on a group of both sharing entities and shared entities (Figure 3.8(c)) in the graph that models the software system being analyzed.

An *associated group* of graph nodes is defined, when two or more source nodes share one or more sink nodes (through direct graph edges). A *source node* is a node where an edge emanates from it. A *sink node* is a node where an edge points to it. In this sense, the whole group of source nodes and sink nodes are denoted as an associated group. In analogy with data mining terminology, we refer to the source nodes as the "*basketset*" and the sink nodes as the "*itemset*".

The entity association between two system entities e_i and e_j in an associated group, denoted as $entAssoc(e_i, e_j)$, is defined as the *maximum* of the association value between e_i and e_j, considering that e_i and e_j may belong to more than one associated groups g_x with a different association value in each group g_x. Formally:

$$entAssoc(e_i, e_j) = max_{g_x} \left(|itemset(g_x)| + w * |basketset(g_x)|\right)$$

where, $0 < w < 1$ is the weight of the sharing entities compared with the shared entities and is discussed later. The entity association is considered as a measure of similarity between two entities in a software system and allows to:

- identify the members of a group of maximally related entities in a system.
- consider the datatypes and variables as members of a group including func-

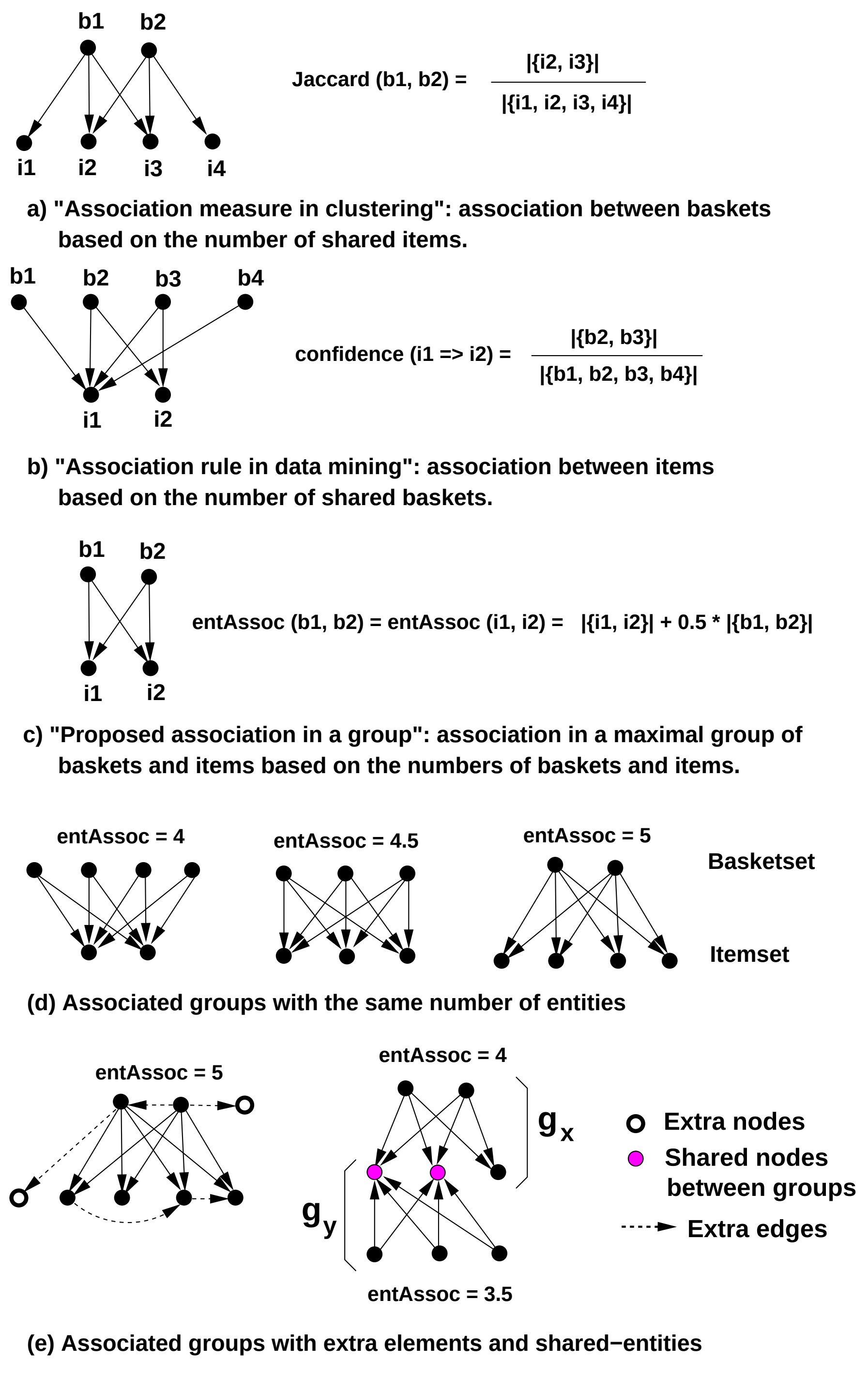

b1 b2
Jaccard (b1, b2) = |{i2, i3}| / |{i1, i2, i3, i4}|
i1 i2 i3 i4
a) "Association measure in clustering": association between baskets based on the number of shared items.
b1 b2 b3 b4
confidence (i1 => i2) = |{b2, b3}| / |{b1, b2, b3, b4}|
i1 i2
b) "Association rule in data mining": association between items based on the number of shared baskets.
b1 b2
entAssoc (b1, b2) = entAssoc (i1, i2) = |{i1, i2}| + 0.5 * |{b1, b2}|
i1 i2
c) "Proposed association in a group": association in a maximal group of baskets and items based on the numbers of baskets and items.
entAssoc = 4
entAssoc = 4.5
entAssoc = 5
Basketset
Itemset
(d) Associated groups with the same number of entities
entAssoc = 5
entAssoc = 4
gx
gy
entAssoc = 3.5
Extra nodes
Shared nodes between groups
Extra edges
(e) Associated groups with extra elements and shared−entities

Figure 3.8: The notion of association in a group as the extension of association in clustering and data mining domains.

tions, as opposed to considering them as attribute-values of functions which cause only the functions to be grouped.

In general, the number of shared entities (itemset) contributes more on closeness of the entities than the number of sharing entities (baskets), if a group of entities are examined for their similarity. We justify this property using a social analogy to software systems:

"*Consider 10 people that eat in the same restaurant and go to the same library. These people can be friends or not. If the number of these people increases from 10 to 20 it does not necessarily increase the level of mutual friendship among them. Now consider the same 10 people and increase the number of their commonalities. For example, suppose they also live in the same building and go to the same club. These people have high likelihood to be friends, since a high number of shared interests is most often an indication of a high level of friendship among people.*"

The lower values of w (close to 0) cause that the $entAssoc(e_i, e_j)$ be less sensitive to the number of sharing entities in an associated group, and vice versa. Based on the empirical results and the above mentioned property, we use a value of $w = 0.5$. The value of $entAssoc(e_i, e_j)$ is a positive *real* number which is not normalized since it measures a property in a single group of entities, not between two groups of entities which allows to normalize the metric (such as Jaccard metric). Hence, its value is not restricted between 0 and 1, instead it depends on the size and form of the group of entities in g_x. A possible way for normalization is to find the *maximum association value* in the system and divide all other values to it. In Figure 3.8(d), three associated groups with the same number of entities along with their association values "*entAssoc*" are shown.

Figure 3.8(e)(left) illustrates the extra nodes and edges that may exist among the nodes and edges of an association group. However, only the solid nodes are the members of the associated group and extra edges do not affect the association value. Figure 3.8(e)(right) illustrates two associated groups g_x and g_y with shared nodes. The grey-color nodes denote the members of both groups with different association values. In this form, the association value of a node is inherited from the group with larger association value.

In the following Section, we define the *source regions* in the source graph to represent a large graph of a software system as a collection of intra-related subgraphs.

3.3.1 Source region

A source-region $G_j^{sr} = (N_j^{sr}, R_j^{sr}, A_j^{sr}, E_j^{sr}, \mu_j^{sr}, \epsilon_j^{sr})$ of a source-graph $G^s = (N^s, R^s, A^s, E^s, \mu^s, \epsilon^s)$ is a subgraph of G^s (i.e., $N_j^{sr} \subseteq N^s, R_j^{sr} \subseteq R^s,\ ..,\ \epsilon_j^{sr} \subseteq \epsilon^s$) and corresponds to a node $n_j \in N^s$. In the source-region G_j^{sr} each node $n_i \neq n_j$ satisfies the association property $entAssoc(n_j, n_i) > 0$ with respect to node n_j. We call n_j the *main-seed* of the source-region G_j^{sr} and use it as the identity of this source-region. Formally:

$$N_j^{sr} = \{n_i \mid n_i \in N^s \quad \wedge \quad \exists n_j \in N^s \ \bullet \ \ entAssoc(n_j, n_i) > 0\} \ \cup \ \{n_j\}$$
$$R_j^{sr} = \{n_s;\ n_t \mid n_s, n_t \in N_j^{sr} \quad \wedge \quad \exists r_k \ \bullet \ r_k = (n_s, n_t) \quad \wedge \quad r_k \in R^s \ \}$$

For a given source-graph $G^s = (N^s, R^s)$ we generate $|N^s|$ source-regions. In general, different source-regions of a source-graph have a number of shared nodes, and some source-regions may be identical.

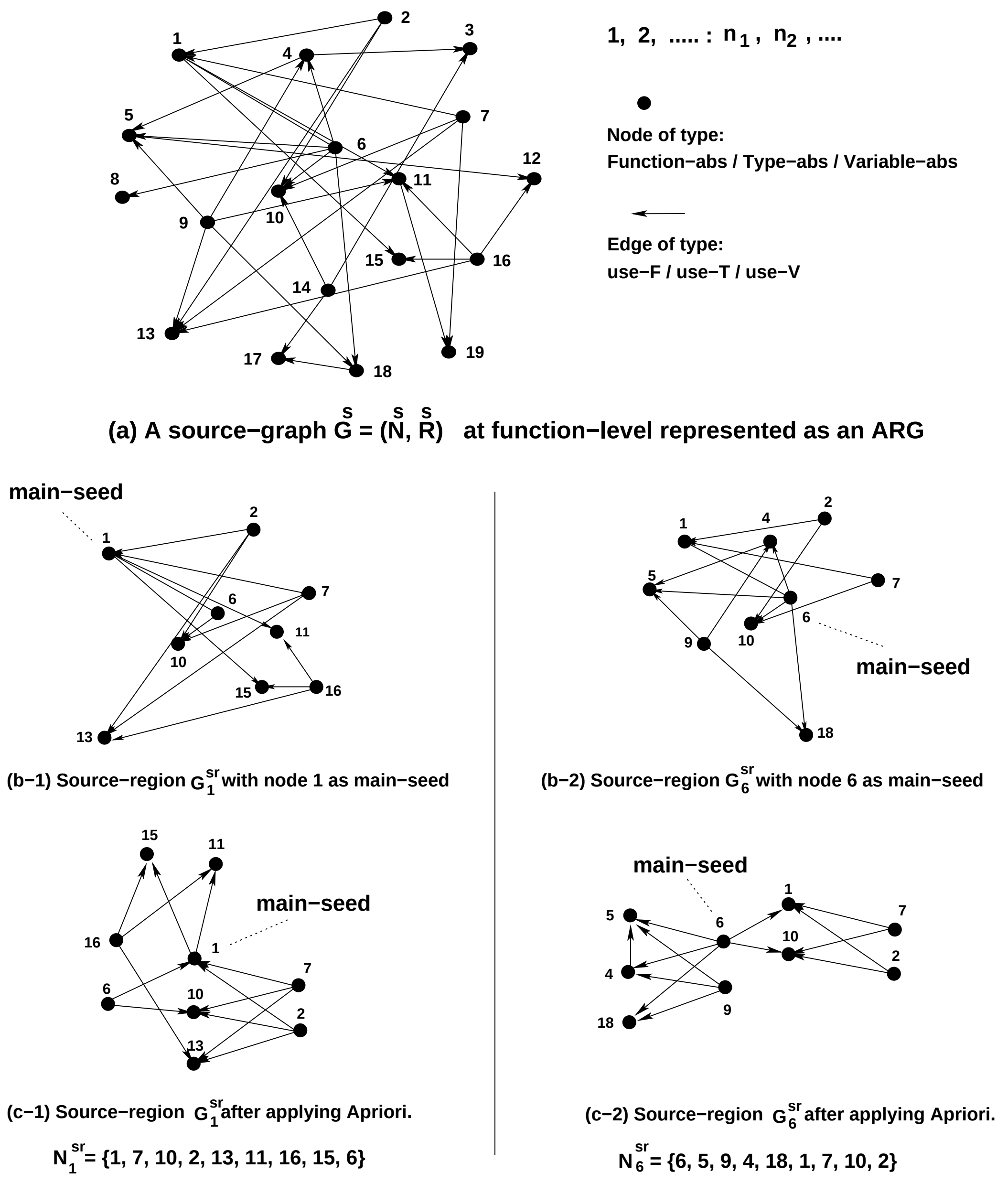

1, 2, ... : n1 , n2 , ...
●
Node of type:
Function−abs / Type−abs / Variable−abs
Edge of type:
use−F / use−T / use−V
(a) A source−graph G^s = (N^s, R^s) at function−level represented as an ARG
main−seed
(b−1) Source−region G^sr_1 with node 1 as main−seed
(b−2) Source−region G^sr_6 with node 6 as main−seed
(c−1) Source−region G^sr_1 after applying Apriori.
N^sr_1 = {1, 7, 10, 2, 13, 11, 16, 15, 6}
(c−2) Source−region G^sr_6 after applying Apriori.
N^sr_6 = {6, 5, 9, 4, 18, 1, 7, 10, 2}

Figure 3.9: Application of data mining on the source-graph $G^s = (N^s, R^s)$ to represent it as a number of source-regions $G^{sr}_j = (N^{sr}_j, R^{sr}_j)$.

For example, Figures 3.9(b-1) and (b-2) represent two source-regions G_1^{sr} and G_6^{sr} of the source-graph G^s, that satisfy the association property, i.e., each node of G_1^{sr} is a member of an associated group with respect to main-seed n_1. However, it is not clear what is the highest association value of each node with regard to main-seed n_1 since each node can be a member of several associated groups, thus different association values in each group g_x with respect to node n_1 are possible. The Apriori algorithm computes all the associated groups in a source-region and allows to determine the maximum association value of each node with respect to the source-region's main-seed, as a measure of similarity. Figures 3.9(c-1) and (c-2) demonstrate the application of the Apriori algorithm on the source-regions in Figure 3.9(b-1) and (b-2). For example, in the source-region G_1^{sr} with node 1 as main-seed in Figure 3.9(c-1), we consider two associated-groups with nodes: 1, 7, 10, 2, 13 with *entAssoc* of 4; and 1, 6, 10, 7, 2 with *entAssoc* of 3.5. The similarity value of node 10 to the main-seed node 1 is 4 and is obtained from the first associated group.

At phase i of the incremental graph matching process, the user may select a main-seed n_j that corresponds to the source-region G_j^{sr} for the current matching phase i. In this context, the main-seed selection can be viewed as a mapping function $g : Integer \rightarrow Integer$ that maps the current matching phase i onto the index j of the selected source-region, i.e., $g(i) = j$. Therefore, a *selected source-region* is represented as $G_{g(i)}^{sr} = (N_{g(i)}^{sr}, R_{g(i)}^{sr})$ which maps to $G_j^{sr} = (N_j^{sr}, R_j^{sr})$.

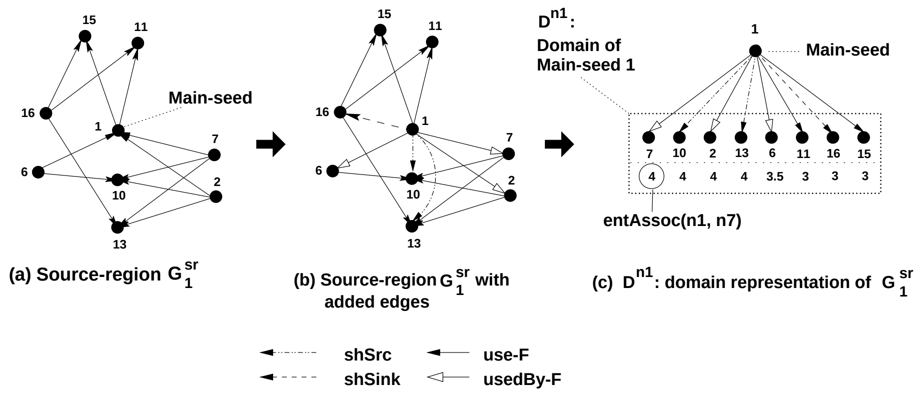


Figure 3.10: The transformation of source-region G_1^{sr} into domain D^{n_1}.

3.3.2 Domain of a node

The *domain* of a node n_j in source-graph G^s, denoted as D^{n_j}, is defined as: *a collection of tuples* (n_d, s_d)*, where the node* n_d *exists in an associated group with node* n_j *and* s_d *is the similarity value between* n_j *and* n_d. The node n_j is called the *main-seed* of domain D^{n_j} and is not included in its domain D^{n_j}. Formally:

$$D^{n_j} = \{n_d;\ s_d \mid n_d \in N_j^{sr} \ \wedge\ \exists n_j \ \bullet\ \textit{main-seed}(n_j, G_j^{sr}) \ \wedge\ s_d = \textit{entAssoc}(n_j, n_d)\}$$

where, $G_j^{sr} = (N_j^{sr}, R_j^{sr})$ is a source-region of the source-graph G^s and the predicate *main-seed*(n_j, G_j^{sr}) indicates that n_j is the main-seed of G_j^{sr}. The nodes of a domain are ranked in descending order according to their similarity values with respect to main-seed n_j, as illustrated in Figure 3.10(c).

A domain D^{n_j} also relates each node of a source region G_j^{sr} with a similarity value that is extracted from the association structure of that source region, and provides means for simple representation of the source-graph as a group of source-regions. Figure 3.10, illustrates the computation of a domain from a source-region, as:

- Two new edge types *shSink* and *shSrc* (denoted as *sharing sink node* and *sharing source node*) are added to each source-region. These types of edges link the main-seed n_j to any unlinked nodes within the source-region, Figure 3.10(b).
- Each edge ending to the main-seed n_j is replaced by an edge starting from the main-seed with inverse relation, e.g., use-F is replaced by usedBy-F.
- A tree with the main-seed n_j as the root and other source-region nodes as the leaves is built, where, the leaves are the domain D^{n_j} of the root node n_j, and each leaf is labeled by its similarity value to the root node, Figure 3.10(c).

3.4 Similarity measure between two groups of entities

In this Section, we proceed to define *group association*, denoted as $groupAssoc(g_i, g_j)$, as a similarity measure between two groups of system entities g_i and g_j based on the similarity between two entities ($entAssoc$) in a graph. This similarity measure allows to extend the scope of the proposed architectural recovery process to file-level granularity and decompose a system into a number of subsystems of files.

In the clustering literature three methods of similarity measures between two groups of entities are commonly used. In the *single linkage* (*complete linkage*) method, the maximum (minimum) similarity value between every pair of entities, one in each group, is considered as the similarity between two groups. The single linkage computes higher similarity value for the groups that are non-compact and isolated, whereas, complete linkage computes higher similarity values for cohesive and compact groups. To avoid the two extremes, the *group average similarity* method defines the similarity between two groups as the average of similarities between all pairs of entities that are made up of one entity from each group [36].

In this thesis, the group average similarity method is adopted to compute the proposed group association metric *groupAssoc*, as follows:

$$groupAssoc(g_i, g_k) = \frac{\sum_{j=1}^{|g_i|} \sum_{m=1}^{|g_k|} sim_{j,m}}{|g_i| + |g_k|} \qquad \text{such that,}$$

$$sim_{j,m} = \begin{cases} 0 & \text{if} \quad n_m \in g_k \quad \wedge \quad (n_m, s_m) \notin D^{n_j} \\ s_m & \text{if} \quad n_m \in g_k \quad \wedge \quad (n_m, s_m) \in D^{n_j} \end{cases}$$

In this equation, the first summation iterates over every entity in group g_i and the second summation iterates over every entity in group g_k in order to add the similarity values $sim_{j,m}$ between every pair of entities, one entity in each group. $Sim_{j,m}$ refers to the similarity value between node n_j in group g_i and node n_m in group g_k. For every entity $n_m \in g_k$ that does not exist in the domain of node n_j (i.e., in D^{n_j}) the similarity value $sim_{j,m}$ between n_j and n_m is zero (i.e., $entAssoc(n_j, n_m) = 0$). Therefore, only those entities in g_k that exist in the domains D^{n_j} (where $n_j \in g_i$) are considered for similarity calculation between two groups. The terms $|g_i|$ and $|g_k|$ denote to the cardinality of each group.

This equation is symmetric with respect to the groups g_i and g_k, i.e.,

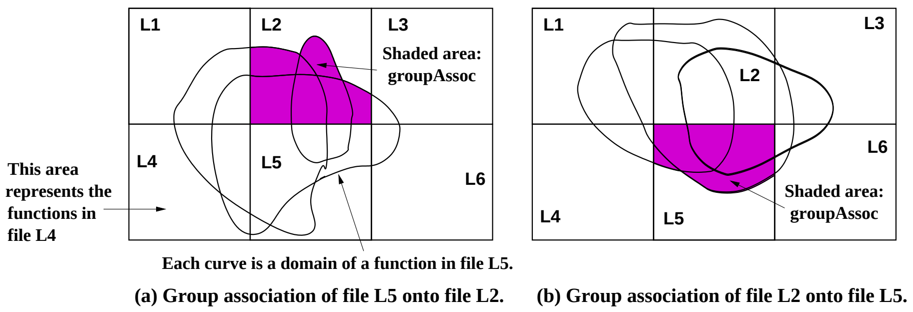

L1
L2
L3
Shaded area: groupAssoc
L4
L5
L6
This area represents the functions in file L4
Each curve is a domain of a function in file L5.
(a) Group association of file L5 onto file L2.
L1
L3
L2
L6
L4
L5
Shaded area: groupAssoc
(b) Group association of file L2 onto file L5.

Figure 3.11: A system of six files representing the group association similarity between two files L2 and L5. Parts (a) and (b) compute the same similarity value.

$groupAssoc(g_i, g_k) = groupAssoc(g_k, g_i)$. This is because in a pair of entities (n_s, n_t) whose similarity value is non-zero, n_s exists in the domain of n_t and n_t exists in the domain of n_s with the same similarity values.

In order to illustrate the group association similarity, we consider *file* as a group of entities and *function* as an entity, where each file contains (defines) a number of functions. In Figure 3.11(a), the domain of each entity (function) in file $L5$ is shown as the area in a closed curve. The Figures 3.11(a) and 3.11(b) illustrate the group association similarities $groupAssoc(L5, L2)$ and $groupAssoc(L2, L5)$ whose values would be the same. The unit for $groupAssoc(g_i, g_k)$ is *"entAssoc per entity"*.

3.5 System representation

Based on the discussion in the previous Sections, we can represent a software system at a higher-level of abstraction in the form of a source-graph G^s along with the collection of domains, which is defined as a two-tuple:

$$system = (G^s, D(N^s))$$

$$where \quad G^s = (N^s, R^s) \quad \wedge \quad D(N^s) = [D^{n_j} \mid j \in [1 .. |N^s|]\ ]$$

$D(N^s)$ is an ordered sequence of entity domains D^{n_j} by the average similarity of each domain, where each domain is a search-space for a module (or subsystem) recovery. In this model, the matching process searches only within the appropriate domains not the whole source graph.

This representation allows to access a source-region G^{sr}_j for the pattern matching process in Chapter 6, where the nodes and edges of the source-region G^{sr}_j are obtained from the domain D^{n_j} and source-graph G^s, respectively.

This system representation has two variations for the types of entities in the domains with respect to two levels of architectural analysis. For the *file-level* analysis, the entities are of types *File-abs, Function-abs, Type-abs, Variable-abs*, and the relations are *cont-R, use-R, imp-R, exp-R*. For the *function-level* analysis, the nodes are simple entities of types *Function-abs, Type-abs, Variable-abs*, and the relations are *use-F, use-T, use-V*.

3.6 Summary

The source-graph G^s is a core data model for the representation of a software system in the proposed architecture recovery approach. Based on the relations among the entities in the source-graph G^s, we defined two similarity metrics between two entities and between two groups of entities (i.e., *entAssoc* and *groupAssoc*, respectively) that use the maximal association property among the group of system entities. The maximal association property is obtained by the application of data mining techniques on the source-graph G^s.

Based on the observation that the size of a typical legacy software system is large and any search or analysis algorithm is intractable for the whole set of system entities, a technique that decomposes the whole search space into a collection of domains based on association property can be of a great value on applying complex analysis algorithms. The search space decomposition, presented in this Chapter, aims to reduce the complexity of the architecture recovery process that will be discussed in Chapter 6. For this work, the software system is presented in the form of a list of domains where each domain will serve as a search-space for the recovery process.

Chapter 4

Architecture query specification

The pattern-based approaches to software architecture recovery first compose a high-level mental model of the system architecture (also known as the conceptual architecture or architectural pattern) using a modeling means such as a query language [56, 55, 79, 48, 39] or a block diagram [38, 78]. In this context, a query language allows the user to compose a query that corresponds to an hypothesis architectural pattern in terms of modules (or subsystems) whose type, size, and interactions form the constraints of the recovery process. Such a query language can query the features that are usually specified by the architecture description languages (ADL) to specify the high level components and connectors in large systems.

This Chapter presents the syntax and the semantics of a query language that we call *Architecture Query Language* (AQL). The Architecture Query Language is based on the concept of architecture description languages and can model a wide range of architectural features and constraints, including: i) constructing the system architecture both at the *file-level* (i.e., partition of the system files into subsystems), and at the *function-level* (i.e., partition of the system functions/datatypes/variables into modules); ii) defining a typed import, export and containment properties of the subsystems and modules; and iii) constraining the composition and interconnection size and type between subsystems or modules. The AQL provides the syntactic and semantic means for the user to define an architectural pattern of the system that can be matched against the artifacts extracted from the source-code. The pattern matching process can compute a sub-optimal match for the architectural pattern and yields a concrete architecture. The details of generating a pattern-graph from an AQL textual query and matching the pattern-graph with the software system source-graph are discussed in Chapters 5 and 6, respectively.

This Chapter is organized as follows: first the notion of system component used in this thesis is defined; second the AQL domain model is discussed; and finally the syntax and semantics of the AQL are defined in detail.

4.1 System component

In this Section, we provide a formal definition for a software system component that was briefly defined in Section 1.1.

Informal definition

We define a software system entity to be a file, function, aggregate/array type, or global variable according to the abstract domain model in Section 3.1.1. A function, aggregate/array type, or global variable is called a "simple entity", while a file is considered to be a "composite entity" as it may contain a set of simple entities. We note that for any given software system, each simple entity must be contained in exactly one file; that is, the set of files form a partitioning of the simple entities. Simple entities may engage in relations, for example: functions may call functions or use aggregate/array types or global variables. For simplicity, we define the *uses* relation to model any such relation between simple entities.

We note that for a given software system, we are interested in modeling only the entities that are defined within the system. In particular, we ignore entities that are defined in external libraries, as well as any relations that they may engage in within the system (e.g., calls to library functions).

We define a "system component" to be a named grouping of system entities. A component may *import* and *export* simple entities; these relations are determined by the relations of the contained simple entities. That is:

- A component imports all of the (simple) entities that are used by its contained (simple) entities, but are not contained by that component.
- A component exports all of its contained (simple) entities that are used by (simple) entities that it does not contain.

There are two kinds of system components:

Module: a component that contains only simple system entities.

Subsystem: a component that contains files (as composite system entities) as well as their contained simple entities, such as functions, datatypes, and variables.

A component may not contain other components. Typically, for a given software system we are concerned with a set of components that forms a partitioning of the software system's entities. We do not mix subsystems and modules; a system partition consists of either all modules or all subsystems.

Formal definition

Let S be a software system. Let N^s denote the set of system entities of S, and let R^s denote the *uses* relation among the simple entities of S.

Let $P(N^s) = \{P_1, ..., P_k\}$ be a partitioning of N^s. For each P_i, we define:

$$P_{i_{imports}} = \{e_1 : N^s \setminus P_i \mid \exists e_2 \in P_i \quad \wedge \quad e_2 \ uses \ e_1\}$$

$$P_{i_{exports}} = \{e_1 : P_i \mid \exists e_2 \in N^s \setminus P_i \quad \wedge \quad e_2 \ uses \ e_1\}$$

Where, $uses = \{use\text{-}F \ \cup \ use\text{-}T \ \cup \ use\text{-}V\}$ corresponds to the relations defined in Section 3.1.1. For each P_i, we can define component C_i as a triple:

$$C_i = \langle P_i, \ P_{i_{imports}}, \ P_{i_{exports}} \rangle$$

We will use these concepts in this dissertation.

4.2 Architecture Query Language (AQL)

The syntactic constructs that define AQL conform with a domain model. An AQL query consists of a number of *abstract components* and *abstract connectors.* Each abstract component is specified as a collection of *placeholders.* The interconnection among the abstract components is established by the means of *abstract connectors,* where an abstract connector is also specified by a number of placeholders. A placeholder is a node in the graph expansion of the AQL query that can be matched

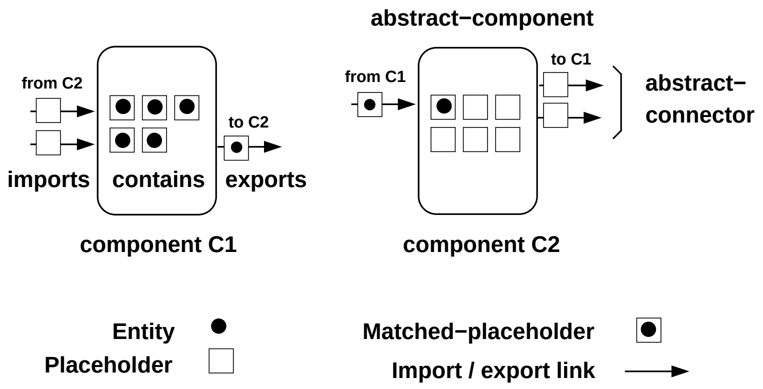


Figure 4.1: The notions of abstract component and abstract connector in AQL, where the placeholders of abstract component C1 have been matched.

(or instantiated) by a system entity during the matching process. The notions of abstract component and abstract connector are illustrated in Figure 4.1.

The user can constrain the minimum and maximum numbers of the matched placeholders as well as the type of the placeholders in the recovered components and connectors by formulating the AQL query. The pattern matching process matches the placeholders with system entities so that the specified constraints in AQL query are satisfied. In the following Sections, the domain model, syntax, and semantics of the AQL are discussed.

4.2.1 Domain model of AQL

The domain model of the AQL is illustrated in Figure 4.2 as an UML class diagram, and the attributes of each class in Figure 4.2 are described in Tables 4.1 and 4.2.

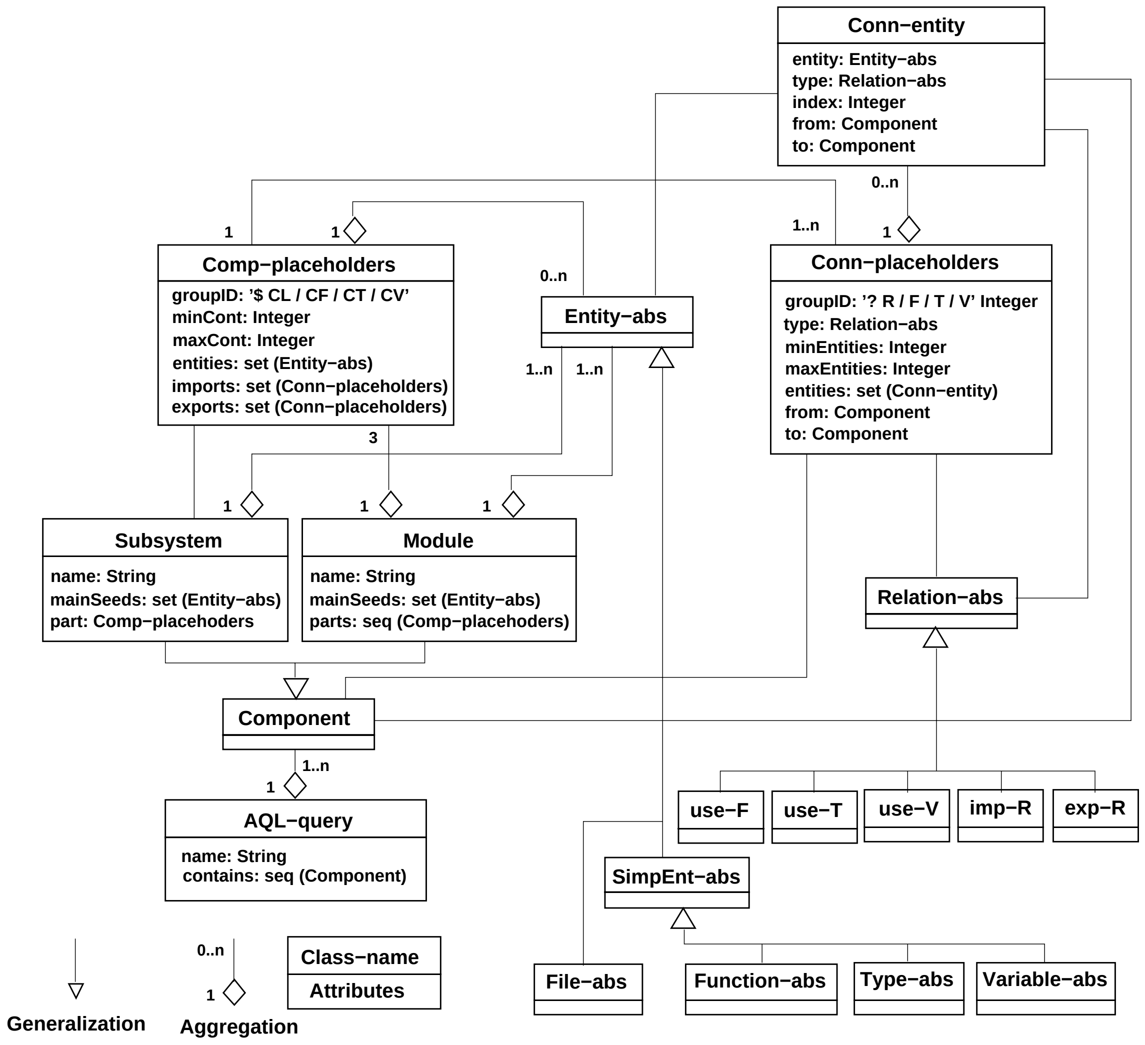


Figure 4.2: The domain model of the proposed Architecture Query Language (AQL). This domain model is used to define the attributes for nodes and edges of the ARG query-graph in Section 5.2.1.

AQL query		
Attribute	*Example*	*Description*
name	Q_y	Unique identifier for each query
contains	$[S_1, S_2, ..]$	list of components (subsystems or modules) contained in the AQL query
Subsystem / Module		
Attribute	*Example*	*Description*
name	S_1 / M_1	Unique identifier for subsystem or module
mainSeeds	{foo}	Entities that specify the source-regions as search space for subsystem or module recovery
part(s)		Specifies the containment constraints, and the relation constraints to other components in the AQL query. Subsystem has only one part for *File-abs.* Module has three parts for *Func-abs, Type-abs, Variable-abs.*
Comp-placeholders		
Attribute	*Example*	*Description*
groupID	\$CL, \$CF, \$CT, \$CV	Identifier for a group of placeholders to be matched with entities that are subtype of *Entity-abs.*
minCont	5	Min number of entities to match with the placeholders.
maxCont	10	Max number of entities to match with the placeholders.
entities		Actual entities that match with the placeholders. Entities of type *File-abs* for subsystem. Entities of a type *Func-abs, Type-abs, Variable-abs* for module.
imports		Specifies the constraints for the set of entities that are imported from other components.
exports		Specifies the constraints for the set of entities that are exported to other components.

Table 4.1: Description of the class attributes in the AQL domain model.

Conn-placeholders		
Attribute	*Example*	*Description*
groupID	?R2, ?F1, ?T4, ?V1	Identifier for a group of placeholders to be matched with simple entities that are imported/exported.
type	*use-F*	The type of relation that must exist between the recovered entities in the destination component and the imported/exported entities.
minEntities	6	Minimum number of entities to be imported/exported.
maxEntities	9	Maximum number of entities to be imported/exported.
entities		Actual imported/exported entities.
from	S_3	The source component that exports the entity.
to	S_2	The destination component that imports the entity.
Conn-entity		
Attribute	*Example*	*Description*
entity	inputKey	An entity of type *Func-abs*, *Type-abs*, or *Variable-abs* that is imported/exported.
type	*use-F*	The relation between this entity and the corresponding entity in the destination component.
index	4	Used to generate a single id from a *groupID* in the class *Conn-placeholder*, e.g., ?R2-4.
from	S_3	The source component that exports the entity.
to	S_2	The destination component that imports the entity.

Table 4.2: Description of the class attributes in the AQL domain model.

4.2.2 Syntax of AQL

The *extended* BNF notation (also known as EBNF) [101] is used for denoting the syntax of the proposed Architecture Query Language. The notation of the EBNF are as follows: i) "::=" means *is defined*; ii) "< >" delimits the non-terminals; iii) "|" means *or*; iv) "[ ]" denotes *optional* syntax; and v) "{ }" denotes *zero or more instances*. Moreover in the syntax of AQL, "( )" denotes a sequence of non-terminals and terminals, "{ }*" is used for zero or more instances, and "{ }$^+$" is used for one or more instances. The syntax of the AQL is presented in Figure 4.3.

4.2.3 AQL query example

A part of an AQL query, consisting of a subsystem S1 of files and its import/export connectors to other subsystems is shown in Figure 4.4.

This AQL fragment is interpreted as follows: after the matching process, the subsystem S1 will IMPORT minimum six and maximum ten resources from subsystem S2 (relating to the matching for "?R1(6 .. 10) S2"); it will EXPORT minimum 10 and maximum 15 resources to subsystem S2 (relating to the matching for "?R3(10 .. 15) S2"); and it will CONTAIN between seven to ten files (by matching the placeholder nodes "$CL(7 .. 10)" including the files *e_edit* and *e_update* that are main seeds). A similar interpretation holds for the sections related to importing from S4 and exporting to S3. The notations ?IR and ?ER in the imports and exports parts denote the unconstrained connectors to other subsystems that are not defined in this AQL query, e.g., importing any number of resources from S3 and exporting any number of resources to S4.

4.2.4 Semantics of AQL

In this Section, we define the semantics of the AQL language.

```
< AQLquery >  ::=  BEGIN-AQL  < directives >  {< component >}+
                     < restOfSys >  END-AQL
< directives >  ::=  $dir_1$ $dir_2$ ... $dir_n$
< component >  ::=  < compType >  :  < compName >  < mainSeeds >
                     < shEx >  < import >  < export >  < contain >
                     < relocate >  END-COMPONENT
< compType >  ::=  SUBSYSTEM | MODULE
< compName >  ::=  < identifier >
< mainSeeds >  ::=  MAIN-SEEDS : ({< srcEntities >}+
                                   | MANUAL-COMPONENT)
< srcEntities >  ::=  {file < identifier > ,}+ |
                (F: {func < identifier > ,}+  T: {type < identifier > ,}+
                 V: {var < identifier > ,}+)
< shEx >  ::=  SHRINK-EXPAND :  E | G | S
< import >  ::=  IMPORTS :  < absLinks >
< export >  ::=  EXPORTS :  < absLinks >
< absLinks >  ::=  (FUNCTIONS: < funcLinks >  TYPES: < typeLinks >
                  VARIABLES: < varLinks >)
                  |  RESOURCES: < rsrcLinks >
< funcLinks >  ::=  func  (?IF | ?EF),
                  {func ?F  < integer >  < intRange >  < compName > ,}*
< typeLinks >  ::=  type  (?IT | ?ET),
                  {type ?T  < integer >  < intRange >  < compName > ,}*
< varLinks >  ::=  var  (?IV | ?EV),
                  {var ?V  < integer >  < intRange >  < compName > ,}*
< rsrcLinks >  ::=  rsrc  (?IR | ?ER),
                  {rsrc ?R  < integer >  < intRange >  < compName > ,}*
< contain >  ::=  CONTAINS:  < contEnts >  |  < contFiles >
< contEnts >  ::=  FUNCTIONS:  func  $CF  < intRange > ,
                              {func  < identifier > ,}+
                    TYPES:  type  $CT  < intRange > ,
                              {type  < identifier > ,}+
                    VARIABLES:  var  $CV  < intRange > ,
                              {var  < identifier > ,}+
< contFiles >  ::= FILES: file $CL < intRange > ,
                                 {file < identifier > ,}+
< relocate >  ::= RELOCATES: (YES: | NO:)
                                {< relocateEnts > TO: < compName > ,}*
< relocateEnts >  ::=  (files | funcs | types | vars) {< identifier > ;}+
< restOfSys >  ::= REST-OF-SYSTEM :
                      IMPORTS: < restShEx >
                      EXPORTS: < restShEx >
                      CONTAINS: < restShEx >
                          CLOSENESS: < restCloseness >
                      DISTRIBUTES:
                          {< restDistribute > ;}*   |
                           ({F : < restDistribure > ;}*
                            {T : < restDistribure > ;}*
                            {V : < restDistribure > ;}*
                           )
< restShEx >  ::=  S | E
< restCloseness >  ::= L: < integer >
                     | (F: < integer > T: < integer > V: < integer >)
< restDistribute >  ::= (files | funcs | types | vars) {< identifier > ,}*
                       TO: ({< compName > ,}+ | ALL)
< intRange >  ::= "(" < integer > .. < integer > ")"
< identifier >  ::= a string of Characters
< integer >  ::= an Integer value
```

Figure 4.3: The syntax of the proposed Architecture Query Language (AQL) described in EBNF notation.

```
SUBSYSTEM: S1
  MAIN-SEEDS:        file  e_edit,  file e_update
  IMPORTS:
    RESOURCES:       rsrc  ?IR,
                     rsrc  ?R1(6 .. 10) S2,
                     rsrc  ?R2(12 .. 20) S4
  EXPORTS:
    RESOURCES:       rsrc  ?ER,
                     rsrc  ?R3(10 .. 15) S2,
                     rsrc  ?R4(1 .. 5) S3
  CONTAINS:
    FILES:           file  $CL(7 .. 10),
                     file  e_edit,  file  e_update
  RELOCATES:         NO:
                     files  e_allign,  u_scale    TO:  S3
END-COMPONENT
```

Figure 4.4: An example of a subsystem in an AQL query.

AQL query

At the *file-level* analysis, an AQL query consists of one or more subsystems and the rest-of-system, where a subsystem is defined as a collection of files and their contained simple entities. At the *function-level* analysis, an AQL query consists of one or more modules and the rest-of-system, where a module is a collection of simple entities such as functions, datatypes, and variables. Each AQL query has a file name, e.g., Q_y, as an identification, and a set of directives dir_x to control the recovery process. The general structure of an AQL query is illustrated in Figure 4.5.

In the following discussion, the semantics of a subsystem and the rest-of-system

```
BEGIN-AQL
        dir_1 ..... dir_n
        SUBSYSTEM (or MODULE) ..... END-COMPONENT
        SUBSYSTEM (or MODULE) ..... END-COMPONENT
                        ..........
        REST-OF-SYSTEM
END-AQL
```

Figure 4.5: The structure of an AQL query.

are defined. Finally the semantics of a module are briefly discussed. The semantics of the AQL constituents are defined in terms of a semantic denotational function "δ" that is:

$$\delta : AQL\ syntax \longrightarrow semantic\ description$$

AQL directives

The AQL directives (dir_1 .... dir_n) control the recovery process by allowing the user to define specific parameters. The directives allow the user to: i) select the type of analysis as file-level or function-level analysis; ii) select the type of entities that are considered for import/export among components; iii) perform non-stop recovery for all components, or stop after each component recovery to allow change of parameters and redo the same component recovery; iv) change the order of component recovery as a requirement for the matching process; v) control the distribution and relocation of entities among the recovered components; vi) merge the abstract components and their abstract connectors to simplify a part of the AQL query; vii) and generate different views of the system such as *control passing, data exchange,* and *data sharing* views [88].

Subsystem

A subsystem S_i in an AQL query Q_y is a five-tuple, as presented below, that is defined between two keywords SUBSYSTEM and END-COMPONENT. A subsystem is an architectural component denoted as a set of source-code files and their contained simple entities, excluding the libraries. A subsystem has a constrained fan-in/fan-out interaction pattern with other subsystems. The structure of a subsystem S_i is illustrated in Figure 4.6.

δ (SUBSYSTEM S_i) $\triangleq$

(MAIN-SEEDS, CONTAINS, IMPORTS, EXPORTS, RELOCATES)

Different parts of a subsystem S_i are defined as follows:

- MAIN-SEEDS: denotes one or more files where each file corresponds to a domain D^{n_j} (defined in Section 3.3.2) as the search space for the selection of the files that would be contained in the subsystem S_i. The collection of the domains for all main-seeds n_j's, denoted as "*domain of the subsystem S_i*" D^{S_i}, constitutes the search space for the matching process.

δ (SUBSYSTEM : S_i

MAIN-SEEDS: file $lname_m$) $\triangleq$

$\{l_m : \textit{File-abs} \mid l_m = Obj(lname_m)^1\}$

[1]Function $Obj(lname_m)$ returns the object of the file name $lname_m$.

```
SUBSYSTEM: $S_i$
    MAIN-SEEDS:         file  $lname_m$,  ...
    IMPORTS:
        RESOURCES:      rsrc   ?IR,
                        rsrc   ?$R_x(min_x$ .. $max_x)$  $S_j$,
                        ....
    EXPORTS:
        RESOURCES:      rsrc   ?ER,
                        rsrc   ?$R_y(min_y$ .. $max_y)$  $S_k$,
                        ....
    CONTAINS:
        FILES:          file  \$CL($min..max$),
                        file  $lname_m$,  file  $lname_s$
                        ....
    RELOCATES:          NO: / YES:
                        files   $lname_r$,  ...  TO:  $S_t$;
                        ...
END-COMPONENT
```

Figure 4.6: An AQL subsystem.

The domain of the subsystem S_i consists of "file, association-value" tuples, as:

$D^{S_i} = \bigcup_{n_j \in \text{MAIN-SEEDS}(S_i)} D^{n_j}$ and $\forall (n_k, s_k) \in D^{S_i} \bullet n_k : \textit{File-abs}$

where, the function MAIN-SEEDS(S_i) returns the set of main-seeds in the subsystem S_i. The keyword "file" above indicates that the main-seed is a file. All the main-seeds of a subsystem are also contained in that subsystem.

- CONTAINS: denotes a collection of system files that are selected from the subsystem's domain D^{S_i} and are related to each other by a binary relation $\mathcal{R}$ such as the association relation. The number of selected files is restricted within a size range (*min, max*). We say that a subsystem contains a group of placeholders with group-id \$CL (as Contains fiLe) and size "*max*" which will be matched with at least "*min*" system files after the recovery process. The placeholders are first matched with the main-seeds and zero or more user-defined fixed files (called *seeds*) from the subsystem's domain D^{S_i}. The CONTAINS part is defined as follows:

  δ (SUBSYSTEM : S_i

  CONTAINS:

  FILES: file \$CL(min, max),

  file $lname_m$,

  file $lname_s$) $\triangleq$

  $$\mathrm{CL}_{S_i} \cup \mathrm{CE}_{S_i} \quad \text{where:}$$

  $$\mathrm{CL}_{S_i} = \{l_m,\ l_s : \textit{File-abs}\} \ \bigcup$$
  $$\{l_k : \textit{File-abs} \mid (l_k, s_k) \in D^{s_i} \ \wedge \ min \leq |\mathrm{CL}_{S_i}| \leq max - 2\} \ \bullet$$
  $$l_m = Obj(lname_m) \ \wedge \ l_s = Obj(lname_s) \ \wedge \ (l_s, s_s) \in D^{s_i};$$

  $$\mathrm{CE}_{S_i} = \{e_t : AnyType \mid AnyType \in \{\textit{Function-abs},\ \textit{Type-abs},\ \textit{Variable-abs}\} \ \wedge$$
  $$\forall e_t,\ \exists l_k \in \mathrm{CL}_{S_i} \ \bullet \ (l_k, e_t) \in \textit{cont-R}\}$$

In the above, $lname_m$ and $lname_s$ are the names of a main-seed file and a seed file; CL_{S_i} and CE_{S_i} are the set of files and the set of simple entities that are contained in the subsystem S_i; and CL_{S_i} satisfies the properties of the *contains* part of a component defined in Section 4.1.

The keywords "FILES" and "file" indicate that only the files of the subsystem S_i that are contained in CL_{S_i} are considered for the size constraint checking against (*min, max*).

- IMPORTS: denotes a set of zero or more simple entities, i.e., functions, datatypes, or variables, that are used (relation *use-R*) by the files of the subject subsystem S_i but are contained (relation *cont-R*) in the files of another subsystem S_j [2]. The identifier ?R_x denotes a group of the imported resources (i.e., simple entities) from the subsystem S_j, where the size of the group is restricted within a size range (min_x, max_x). Two characteristics of the imported entities include:
  - Each entity is imported once to the same subsystem, hence, an already imported entity does not match with another imported placeholder to the same subsystem.
  - All other imported entities from the other subsystems that are not defined in the IMPORTS part of the subject subsystem S_i, are identified as ?IR. There is no constraint on the number of the imported entities in the ?IR group.

[2]Since the containment relation is transitive, the subsystem S_j contains the simple entities that are contained in its files.

The IMPORTS part is defined as follows:

δ (SUBSYSTEM : S_i

IMPORTS:

RESOURCES: rsrc ?IR,

rsrc ?R$_x(min_x, max_x)$ S_j) $\triangleq$

$$\mathrm{R}_x \cup \bigcup_{k=1,\ k\neq i,j}^{|subsystems|} \mathrm{IR}_{S_k} \qquad \text{where:}$$

$$\mathrm{R}_x = \{R : \textit{Entity-abs} \mid \exists L, L' : \textit{File-abs} \bullet L \in \mathrm{CONTAINS}(S_i)^3 \wedge L' \in \mathrm{CONTAINS}(S_j) \wedge (L,R) \in \textit{use-R} \wedge (L',R) \in \textit{cont-R} \wedge i \neq j \wedge min_x \leq |\mathrm{R}_x| \leq max_x\}$$

$$\mathrm{IR}_{S_k} = \{R : \textit{Entity-abs} \mid \exists L, L' : \textit{File-abs} \bullet L \in \mathrm{CONTAINS}(S_i) \wedge L' \in \mathrm{CONTAINS}(S_k) \wedge (L,R) \in \textit{use-R} \wedge (L',R) \in \textit{cont-R}\}$$

The keywords "RESOURCES" and "rsrc" (abbreviation of *resource*) indicate that the entities in this group are simple entities of types *Function-abs, Type-abs*, or *Variable-abs.*

- EXPORTS: denotes a set of zero or more simple entities (i.e., functions, datatypes, variables) that are contained (relation *cont-R*) in the files of the subject subsystem S_i and are used (relation *use-R*) by the files of the other subsystems, e.g., S_k. The identifier ?R$_y$ denotes a group of exported resources

[3]Function CONTAINS(S_i) returns the system files that are contained in the subsystem S_i.

(i.e., simple entities) to the subsystem S_k, where the size of the group is restricted within a size range $(min_y,\ max_y)$.

All other exported entities to the other subsystems that are not defined in the EXPORTS part of the subject subsystem S_i, are identified as ?ER. There is no constraint on the number of the exported entities in the ?ER group. The EXPORTS part is defined as follows:

δ (SUBSYSTEM : S_i

    EXPORTS:

        RESOURCES: rsrc ?ER,

            rsrc ?$\mathrm{R}_y(min_y, max_y)\ S_k$) $\triangleq$

$\mathrm{R}_y \quad \cup \quad \bigcup_{p=1,\ p\neq i,k}^{|subsystems|} \quad \mathrm{ER}_{S_p}$ where:

$$\mathrm{R}_y = \{R : \textit{Entity-abs} \ \mid \ \exists L, L' : \textit{File-abs} \bullet \ L \in \mathrm{CONTAINS}(S_i) \ \wedge$$
$$L' \in \mathrm{CONTAINS}(S_k) \ \wedge \ (L, R) \in \textit{cont-R} \ \wedge \ (L', R) \in \textit{use-R} \ \wedge$$
$$i \neq k \ \wedge \ min_y \leq |\mathrm{R}_y| \leq max_y\}$$
$$\mathrm{ER}_{S_p} = \{R : \textit{Entity-abs} \ \mid \ \exists L, L' : \textit{File-abs} \bullet \ L \in \mathrm{CONTAINS}(S_i) \ \wedge$$
$$L' \in \mathrm{CONTAINS}(S_p) \ \wedge \ (L, R) \in \textit{cont-R} \ \wedge \ (L', R) \in \textit{use-R}\}$$

The keywords "RESOURCES" and "rsrc" (abbreviation of *resource*) indicate that the entities in this group are simple entities of types *Function-abs, Type-abs*, or *Variable-abs.*

- RELOCATES: denotes zero or more files in the subject subsystem S_i that are selectively assigned to a destination subsystem, e.g., S_t. The relocation operation is effective after the recovery process is terminated, all subsystems are recovered, and the size constraints are satisfied. The file relocation is performed in order to possibly improve the size or quality of the recovered subsystems, however it may violate the size and/or interaction constraints of the subsystems. The RELOCATES part is defined as follows:

  δ (SUBSYSTEM : S_i
  RELOCATES: YES:
  files $lname_r$ TO: S_t) $\triangleq$

  if $\exists l_r : \mathit{File\text{-}abs} \mid l_r = Obj(lname_r) \wedge l_r \in \text{CONTAINS}(S_i)$ then
  $\text{CONTAINS}(S_i)'$[4] $= \text{CONTAINS}(S_i)\ \mathit{less}\ l_r \wedge$
  $\text{CONTAINS}(S_t)' = \text{CONTAINS}(S_t)\ \mathit{with}\ l_r$

  The keywords "NO / YES" are used to turn-off/on the relocation operation, and the keyword "files" indicates that only the files of the subsystem may be relocated.

Manual-subsystem

A *manual-subsystem* is a subsystem that all of its contained files have been selected

[4]Function $\text{CONTAINS}(S_i)'$ returns the contained files in subsystem S_i after the relocation operation.

by the user. There are no constraints on the selection of the files and at the end of the recovery process all the recovered subsystems will become mutually disjoint against every manual-subsystem. The files of a manual-subsystem may be chosen from different domains. The syntax of a manual subsystem is similar to the syntax of a constrained subsystem with the following differences: i) The keyword <<MANUAL-COMPONENT>> is used instead of the list of main-seed files; ii) all the lines that are used for constraining the sizes in the parts IMPORTS, EXPORTS, and CONTAINS, are removed. The CONTAINS part of a manual-subsystem is defined as follows:

$$\begin{aligned}
&\delta\ (\text{SUBSYSTEM}:\ S_{manual_j}\\
&\qquad \text{CONTAINS:}\\
&\qquad\qquad \text{FILES:}\quad \text{file}\ \ lname_s,\\
&\qquad\qquad\qquad\quad\ \ \text{file}\ \ lname_t\ \ )\quad \triangleq
\end{aligned}$$

$$\begin{aligned}
&\{l_s,\ l_t : \textit{File-abs}\}\ \mid\ l_s = Obj(lname_s)\ \ \wedge\ \ l_t = Obj(lname_t)\ \ \wedge\\
&\qquad\qquad \forall S_{i_{i\neq \text{manual}_j}} \in \textit{AQL-query}\ \ \bullet\ \ \{l_s,\ l_t\} \cap \text{CONTAINS}(S_i)\ \ = \phi
\end{aligned}$$

Rest of system

The *rest-of-system* contains the remaining of the system files, after termination of the recovery process. The syntax of the rest-of-system in subsystem recovery is shown in Figure 4.7.

In the proposed incremental architectural recovery process, initially all the files are contained in the rest-of-system, and during the recovery process the files are

```
REST-OF-SYSTEM:
    IMPORTS:     S
    EXPORTS:     S
    CONTAINS:    E

    DISTRIBUTES:
      files $lname_d$, ...  TO:  $S_j$, $S_k$, ...
          ......
```

Figure 4.7: Rest-of-system in AQL query.

removed from the rest-of-system to match with the expanded form of the AQL query. After the termination of the constraint-based recovery process the user can distribute a part of the files in the rest-of-system among the recovered subsystems. All the remaining files in the rest-of-system are ranked based on their average closeness values to every recovered subsystem in the AQL query. This allows the user to select different groups of files from the ranked list and distribute them among the subsystems based on their closeness values.

The semantics of IMPORTS and EXPORTS parts of the rest-of-system are similar to ?IR and ?ER parts of the IMPORTS and EXPORTS defined earlier. The CONTAINS part denotes the remaining files in the system after the recovery process or the distribution operation. To simplify the representation of a large number of imported, exported, or contained entities in the rest-of-system, the expand (E) or shrink (S) is used to switch between detailed representation of the entities or just the quantities of the entities in these parts, respectively. The CONTAINS part and DISTRIBUTES parts of the rest-of-system are defined as follows:

δ (REST-OF-SYSTEM)

CONTAINS: E) $\triangleq$

$\{l_k : \textit{File-abs} \mid \forall S_j \in \text{AQL query} \bullet l_k \notin \text{CONTAINS}(S_j)\}$

δ (REST-OF-SYSTEM)

DISTRIBUTES:

files $lname_d$ TO: S_j, S_k) $\triangleq$

$\exists l_d : \textit{File-abs} \bullet l_d = Obj(lname_d) \wedge$

CONTAINS(REST-OF-SYSTEM)' = CONTAINS(REST-OF-SYSTEM) $\textit{less } l_d \wedge$

if $\textit{average-closeness}(l_d, S_j) > \textit{average-closeness}(l_d, S_k)$ then

$\text{CONTAINS}(S_j)' = \text{CONTAINS}(S_j) \ \textit{with} \ l_d$

else

$\text{CONTAINS}(S_k)' = \text{CONTAINS}(S_k) \ \textit{with} \ l_d$

Module

A module M_i in an AQL query is defined between two keywords MODULE and END-COMPONENT and refers to the analysis performed at the function-level. A module is an architectural component consisting of three groups of simple entities of types *Function-abs*, *Type-abs*, and *Variable-abs* that are collected based on a binary relation $\mathcal{R}$. The sizes of different parts of a module M_i and its interaction with other modules are constrained by the AQL statements. A complete module

```
MODULE: $M_i$
    MAIN-SEEDS:         F: func ... T: type ... V: var ...
    IMPORTS:
        FUNCTIONS:      func  ?IF,
                        func  ?$F_u(a_u$ .. $b_u)$  $M_\alpha$,   func ...
        TYPES:          type  ?IT,
                        type  ?$T_v(c_v$ .. $d_v)$  $M_\beta$,   type ...
        VARIABLES:      var   ?IV,
                        var   ?$V_w(e_w$ .. $f_w)$  $M_\gamma$,   var ...
    EXPORTS:
        FUNCTIONS:      func  ?EF,
                        func  ?$F_x(g_x$ .. $h_x)$  $M_\delta$,   func ...
        TYPES:          type  ?ET,
                        type  ?$T_y(k_y$ .. $l_y)$  $M_\epsilon$,   type ...
        VARIABLES:      var   ?EV,
                        var   ?$V_z(m_z$ .. $n_z)$  $M_\eta$,   var ...
    CONTAINS:
        FUNCTIONS:      func   $CF(o .. p),
                        func  ...,  func  ...
        TYPES:          type   $CT(q .. r),
                        type  ...,  type  ...
        VARIABLES:      var     $CV(s .. t),
                        var  ...,  var  ...
    RELOCATES:          NO: / YES:
                        funcs   ....  TO:  $M_j$;    types ...;    vars ...
END-COMPONENT
```

Figure 4.8: An AQL module.

specification example in an AQL query is illustrated in Figure 4.8. However, in most applications only a subset of these parts are defined. The semantics of AQL constructs for modules is the same as for subsystems with the only difference that files are replaced by functions, datatypes, or variables.

4.3 Query generation

An important step in a pattern based architectural recovery process is the generation of the initial pattern. We propose the following methods to generate a pattern in an AQL query: i) analyzing the association relation among the system entities [88]; ii) applying a clustering technique [89]; iii) comparing the source code of the system with its reference architecture; or iv) using the available system architecture document or consulting with the system developers. The objective in any of these methods is to extract small groups of system entities which represent the core functionality of the modules (or subsystems) in the system architecture. These groups are then used to generate an initial query-graph represented by an AQL query. The details of the query generation will be discussed in Chapter 8.

4.4 Summary

In this Chapter, the syntax and the semantics of the Architecture Query Language (AQL) were presented. The AQL allows to specify a model of the conceptual architecture of a software system that is used as a pattern in an architectural recovery process. More specifically, the generated pattern is defined in terms of abstract components (subsystems or modules) whose interactions can be constrained. The pattern generation has an exploratory and incremental nature that is performed as a part of the pattern matching process. The following Chapters will discuss how the proposed architectural pattern, as an AQL query, is used to incrementally recover the architecture of a software system.

Chapter 5

Pattern graph generation

In chapter 4 we defined the AQL language that allows to specify the high-level architecture of a software system in terms of abstract components and abstract connectors as an AQL query. The next step would be to transform the textual specification of an AQL query into a graph representation (called *query-graph*) and derive a collection of graphs that are required for the graph matching algorithm to be discussed in chapter 6.

In this chapter, first, the graph of a software system and its regions are summarized from chapter 3. Second, the query-graph is defined. Finally, the group of graphs that are generated from the query-graph during the iterative matching process are defined. The generated graphs are related by recursive graph algebraic equations.

5.1 Graphs based on software system

Below, a summary of the concepts of a source-graph and a source-region initially presented in chapter 3 are given. The source-graph is the central data model in the proposed graph pattern matching approach.

Source graph: the source-graph G^S is an attributed relational graph that models the software system under analysis. The nodes (n_j) represent files, functions, datatypes, and variables. The edges (r_y) represent *call* and *use* relationships. The nodes and edges comply with the specific domain model defined for architectural analysis in chapter 3. The source-graph is denoted as:

$$G^s = (N^s, R^s)$$

Source region: the source-region G_j^{sr} is a subgraph of the source-graph G^s that corresponds to a node n_j of G^s, where n_j is called the main-seed of G_j^{sr}. In G_j^{sr} every node $n_k \neq n_j$ satisfies the association property with respect to node n_j. From the source-graph $G^s = (N^s, R^s)$ a collection of $|N^s|$ source-regions are generated, one for each node n_j in the source-graph. The details of the source-region G_j^{sr} have been presented in Section 3.3.1. At matching phase i the user would select a main-seed n_j (where $j = g(i)$) that assigns the source-region G_j^{sr} for the matching process. Therefore, a "*selected source-region*" at matching phase i is represented as:

$$G_j^{sr} = \quad G_{g(i)}^{sr} = (N_{g(i)}^{sr}, R_{g(i)}^{sr})$$

5.2 Graphs based on AQL pattern

In order to model the architectural recovery as a graph pattern matching process, six graphs are defined. The first graph, denoted as *query-graph* G^q, is the product of mapping the textual representation of an AQL query onto an attributed relational graph (ARG). The other five graphs include: i) the *pattern-region* G_i^{pr} that is produced from the expansion of the i'th node in the query-graph G^q; ii) the *matched-region* G_i^{mr} that denotes the result of matching the pattern-region G_i^{pr} with a source-region $G_{g(i)}^{sr}$ at phase i of the matching process; iii) the *pattern-graph* G_i^p that denotes the incremental expansion of a part of the query-graph; iv) the *input-graph* G_i^I that denotes the graph to be matched against the pattern-graph G_i^p; and v) the *matched-graph* G_i^m that denotes the result of matching the pattern-graph G_i^p and input-graph G_i^I . The details of the pattern matching process will be discussed in chapter 6.

5.2.1 Query graph

The *query-graph* $G^q = (N^q, R^q)$ is a multigraph ARG with composite nodes (denoted as *query-nodes*) and composite edges (denoted as *query-edges*). The formal definition of the query-graph G^q is presented in Appendix B. The query-graph G^q can be directly derived from an AQL query, as illustrated in Figure 5.1, where the attributes of the *query-nodes* and *query-edges* conform with the domain model of the Architecture Query Language in Section 4.2.1. These attributes are filled with attribute values that are obtained from the corresponding AQL query text. Each query-node qn_i models an AQL component (module or subsystem), and each query-edge qr_k models a collection of import/export relations between two AQL

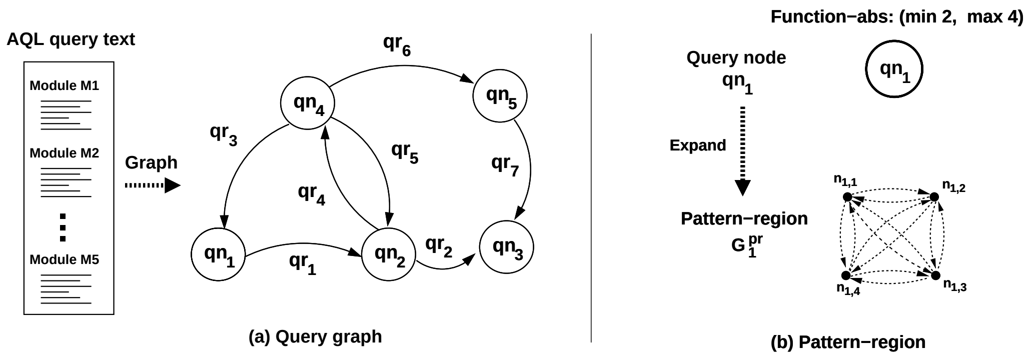


Figure 5.1: (a) Generation of a query-graph with 5 composite nodes from the AQL query text, where the edges represent import/export of resources. (b) Expansion of a query-node qn_1 into pattern-region G_1^{pr}.

components. A query-graph is expanded during the incremental graph matching phases to generate a pattern-graph, as follows: each query-node is expanded into a *pattern-region*, and each query-edge is expanded into a number of *edge-bundles*. Pattern-region, edge-bundles, and pattern-graph are defined below.

In the following Sections, the different graphs that are generated from a query-graph $G^q = (N^q, R^q)$ will be discussed.

5.2.2 Pattern region

The *pattern-region* at matching phase i is an ARG $G_i^{pr} = (N_i^{pr}, R_i^{pr})$ that is generated by expanding the composite node qn_i of the query-graph G^q. The cardinality of the expanded nodes, denoted as *placeholder-nodes* $n_{i,j}$ is the maximum number of entities specified for the i'th component in the AQL query. The placeholder-nodes and their in-between edges are to be matched against the nodes and edges of a subgraph of a selected source-region $G_{g(i)}^{sr}$. Figure 5.1(b) illustrates the expansion of the query-node qn_1 into pattern-region G^{pr}_1. The rationale for such a pattern-region is to create a fully-connected graph for any given subset of nodes in G^{pr}_i, in terms of functions that call each other and functions that all use the same group of datatypes and variables. Since this is the strongest possible connection pattern among the software system entities the approximate matching process will match only a subset of such edges in the pattern-region. The formal definition of pattern-region G^{pr}_i is presented in Appendix B.

5.2.3 Matched region

The *matched-region* at matching phase i is an ARG $G^{mr}_i = (N^{mr}_i, R^{mr}_i)$ which is the result of matching a pattern-region G^{pr}_i with a source-region $G^{sr}_{g(i)}$ at phase i, such that the AQL query import/export link constraints from phase 1 to $i-1$ have already been satisfied. We refer to G^{mr}_i as the "*recovered subsystem i* " or the "*recovered module i* " to denote a subsystem or module of the recovered system architecture at phase i. Figure 5.2(a) illustrates a matched-region G^{mr}_u whose nodes and edges have been matched at phase u $(u < i)$.

5.2.4 Graph connectors and graph summations

This Section defines the concepts that are used for specifying the pattern-graph and the input-graph. The formal definitions of these concepts are presented in Appendix B.

Edge-bundle: is a group of edges that correspond to the partial expansion of a query-edge qr_k in query-graph G^q at matching phase i. An edge-bundle

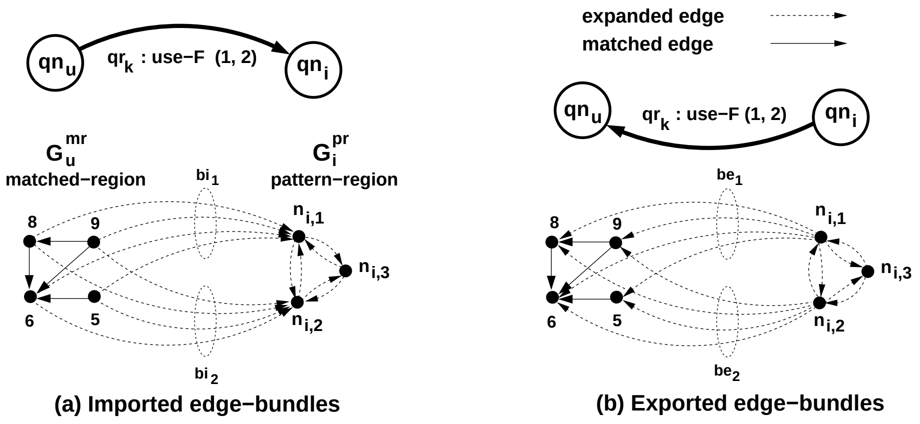


Figure 5.2: Two sets of imported and exported edge-bundles with reference to the pattern-region G_i^{pr}.

connects every node in a matched-region G_u^{mr} $(u < i)$ to one node (either a *sink-node* or a *source-node*) in the pattern-region G_i^{pr}. The types of the connected nodes in both sides of the edge-bundle must conform with the type of the edge-bundle. Figures 5.2(a) and 5.2(b) illustrate two sets of imported and exported edge-bundles for edge-type *use-F*. Since the maximum cardinality of the imported or exported edges is 2 (defined by the query-edge qr_k), two edge-bundles are generated in either case of the example in Figure 5.2.

Connector-edges: are 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-region G_u^{mr} and the source-region $G_{g(i)}^{sr}$ at phase i are denoted by $\mathcal{R}_i^{mr_u \leftrightarrow sr_i}$, and the edge-bundles between a matched-region G_u^{mr} and the pattern-region G_i^{pr} at phase i are denoted by $\mathcal{R}_i^{mr_u \leftrightarrow pr_i}$. Where there is no ambiguity, the indices of the connected graphs can be omitted, as: $\mathcal{R}_i^{mr_u \leftrightarrow sr}$ and $\mathcal{R}_i^{mr_u \leftrightarrow pr}$ for the above connector-edges, respectively.

Graph summation: the binary operator sum "+" is defined in order to compose a graph in terms of its constituent subgraphs. For this thesis the definition of graph summation found in [111] is adopted, where the graph summation is defined as: "*if two graphs G_1 and G_2 are connected (as graph G), then $G_1 + G_2$ is a disconnected graph with two components G_1 and G_2*" as shown in Figure 5.3(a). We also use the binary operator o-plus "$\oplus$" to represent the summation of a graph and a set of edges that yield a new graph as illustrated in Figure 5.3(b).

We can use the graph summation operators "+" and "$\oplus$" to compose different graphs. If two graphs G_1 and G_2 (possibly with shared nodes) are connected via a set of connector-edges $\mathcal{R}^{G_1 \leftrightarrow G_2}$, then we can define G as:

$$G = (G_1 \oplus \mathcal{R}^{G_1 \leftrightarrow G_2}) + G_2 \qquad \text{or}$$
$$G = G_1 + (\mathcal{R}^{G_1 \leftrightarrow G_2} \oplus G_2)$$

Figures 5.3(a) and (b) illustrate two examples of graph summation using "+" and "$\oplus$". These notations are used to provide an equational form for the matching process.

5.2.5 Matched graph

The *matched-graph* at phase i is an ARG $G_i^m = (N_i^m, R_i^m)$ that is a subgraph of the source-graph G^s. The matched-graph G_i^m is the result of the matching process applied on two graphs namely the *pattern-graph* G_i^p and the *input-graph* G_i^I. The

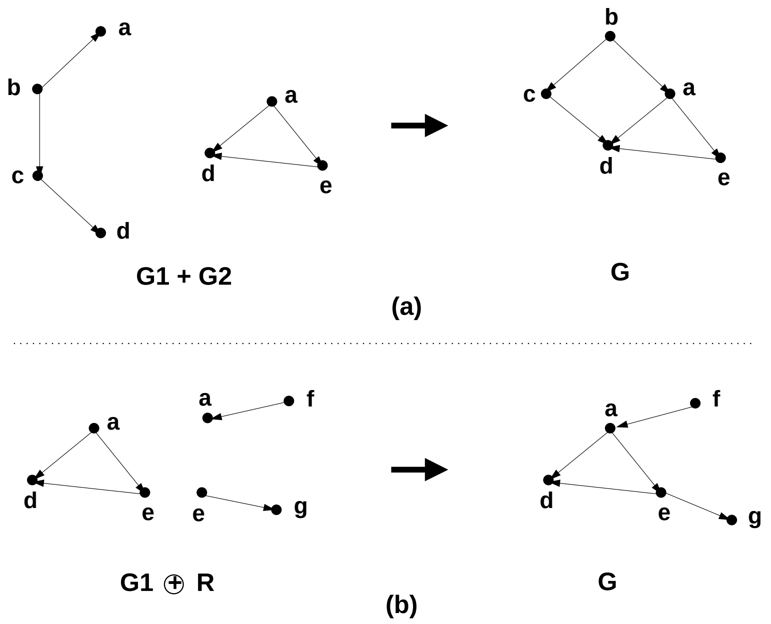


Figure 5.3: (a) The notion of graph summation operator "+" that allows to compose two graphs $G1$ and $G2$ and represent them as a connected graph G. (b) The notion of graph-and-edge summation operator "$\oplus$" that allows to compose a graph $G1$ and a set of edges $R = \{(f, a),\ (e, g)\}$ and represent them as a graph G.

matched-graph is represented as:

$$G_i^m = match(G_i^p, G_i^I) \ = \ G_{i-1}^m \ + \ (\mathcal{R}_i^{m \leftrightarrow mr_i} \ \oplus \ G_i^{mr})$$

where G_i^p and G_i^I are defined in the following Sections. The matched-graph at phase i of the matching process can be represented using the graph summation notations, as a collection of matched-regions and their corresponding connector-edges, such as:

$$G_i^m = (G_1^{mr} \oplus \mathcal{R}_1^{mr^* \leftrightarrow mr_1}) \;+\; .... \;+\; (G_i^{mr} \oplus \mathcal{R}_i^{mr^* \leftrightarrow mr_i}) \qquad \text{that yields}$$

$$G_i^m = \sum_{u=1}^{i} \; (G_u^{mr} \oplus \mathcal{R}_u^{mr^* \leftrightarrow mr_u})$$

where, G_u^{mr} is a matched-region at phase $u \leq i$, and $\mathcal{R}_u^{mr^* \leftrightarrow mr_u}$ is the group of connector-edges between G_u^{mr} and every other matched-region G_j^{mr} such that $1 \leq j \leq i \;\; \wedge \;\; j \neq u$. In other words, the connector-edges $\mathcal{R}_u^{mr^* \leftrightarrow mr_u}$ constitute *all* the import/export links from/to every recovered subsystem S_j or module M_j.

Such a representation provides a "modular view" of the matched-graph G_i^m that allows us to access an individual constituent module G_u^{mr} and its import/export links $\mathcal{R}_u^{mr^* \leftrightarrow mr_u}$ within the matched-graph from the first match to the last match at phase i by indexing. This indexing is necessary for generating the pattern-graph discussed bellow. As mentioned previously, the matched-graph G_i^m can also be represented as a *recursive equation* using the graph summation notations, as shown in Figure 5.4.

$$G_i^m = G_{i-1}^m \;+\; (\mathcal{R}_i^{m \leftrightarrow mr_i} \;\; \oplus \;\; G_i^{mr})$$

5.2.6 Pattern-graph

The *Pattern-graph* at phase i is an ARG $G_i^p \;=\; (N_i^p, R_i^p)$ that is generated by incrementally expanding the query-graph G^q at different graph matching phases. At each phase i only those query-edges that exist between the current query-node qn_i and the previously matched query-nodes qn_1 to qn_{i-1} are expanded. The pattern-

$$G_i^m = \sum_{u=1}^{i} (G_u^{mr} \oplus \mathcal{R}_u^{mr^* \leftrightarrow mr_u})$$

$G_0^m = \phi$ % No matched-graph exists at phase 0.
$G_1^m = G_1^{mr}$ % No connector-edges $\mathcal{R}_1^{\phi \leftrightarrow mr_1}$ exist at phase 1

$G_2^m = (G_1^{mr} \oplus \mathcal{R}_1^{\phi \leftrightarrow mr_1}) + (G_2^{mr} \oplus \mathcal{R}_2^{mr^* \leftrightarrow mr_2})$ % $\mathcal{R}_2^{mr^* \leftrightarrow mr_2} = \mathcal{R}_2^{m \leftrightarrow mr_2}$

$G_2^m = G_1^m + (\mathcal{R}_2^{m \leftrightarrow mr_2} \oplus G_2^{mr})$
....

$G_i^m = G_{i-1}^m + (\mathcal{R}_i^{m \leftrightarrow mr_i} \oplus G_i^{mr})$

Figure 5.4: Representing the matched-graph G_i^m as a recursive equation.

graph is represented using the graph summation, as:

$$G_i^p = G_{i-1}^m \ + \ (\mathcal{R}_i^{m \leftrightarrow pr_i} \ \oplus \ G_i^{pr})$$

where $G_0^m = \phi$. The above equation is interpreted as: at phase i the pattern-graph G_i^p is the sum of the matched-graph at previous phase G_{i-1}^m, pattern-region at current phase G_i^{pr}, and the imported/exported edge-bundles $\mathcal{R}_i^{m \leftrightarrow pr}$ between the pattern-region and the matched-regions inside G_{i-1}^m.

Figure 5.5 illustrates an example of pattern-graph generation. In Figure 5.5(a) a query-graph with 5 query-nodes is shown. In Figure 5.5(b) the pattern-graph is expanded at phase 4, where the edge-bundles in $\mathcal{R}_4^{m \leftrightarrow pr_4}$ connect the pattern-region G_4^{pr} to the matched-regions at phases 1 and 2. Figure 5.5(c) illustrates the expansion of the same pattern-graph at phase 5, where the pattern-graph at phase 4 G_4^{pr} has

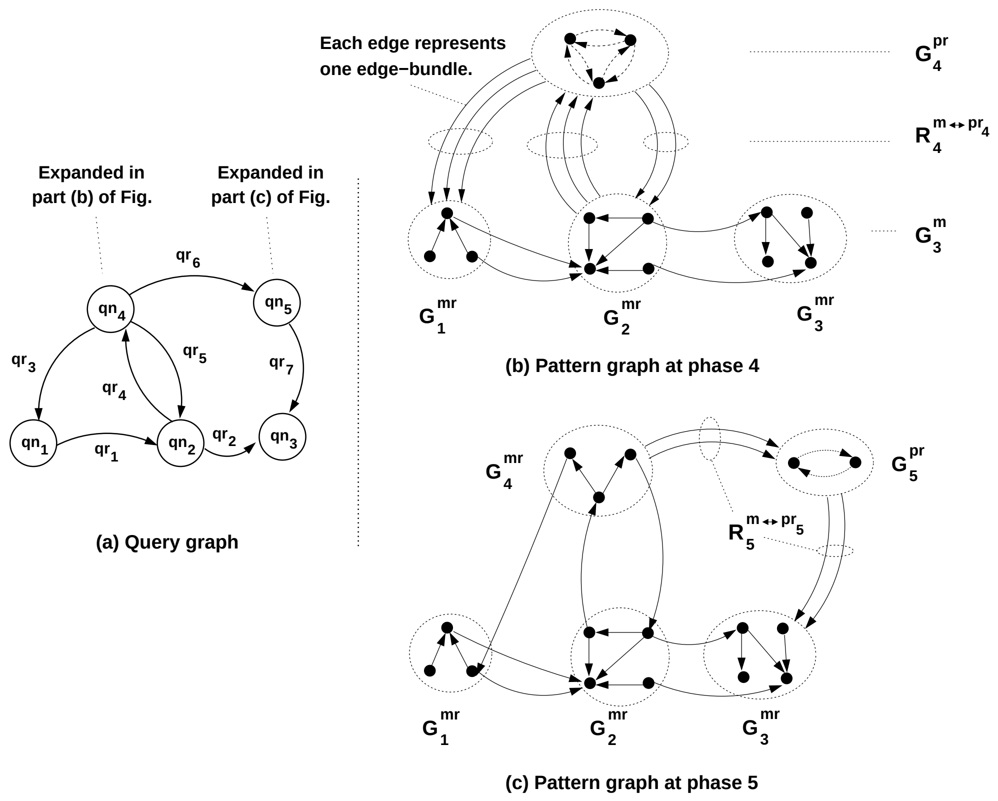

Expanded in part (b) of Fig.
Expanded in part (c) of Fig.
qn_1 qn_2 qn_3 qn_4 qn_5
qr_1 qr_2 qr_3 qr_4 qr_5 qr_6 qr_7
(a) Query graph
Each edge represents one edge−bundle.
G_4^{pr}
$R_4^{m \leftrightarrow pr_4}$
G_3^m
G_1^{mr} G_2^{mr} G_3^{mr}
(b) Pattern graph at phase 4
G_4^{mr} G_5^{pr}
$R_5^{m \leftrightarrow pr_5}$
G_1^{mr} G_2^{mr} G_3^{mr}
(c) Pattern graph at phase 5

Figure 5.5: (a) *query-graph* with five query-nodes. (b) Edge-bundles corresponding to the query-edges qr_3, qr_4, and qr_5 connect the *pattern-region* G_4^{pr} to the *matched-regions* G_1^{mr} and G_2^{mr}. (c) The pattern-graph at phase 5.

already been matched. The summation representation of the matched-graph G_i^m allows such access to the matched-regions at previous phases.

Figures 5.6(a) and (b) illustrate the detailed generation of a pattern graph G_4^p from the query-graph G^q at phase 4. The procedure for expanding a query-edge qr_k into edge-bundles generates p (max size qr_k) edge-bundles with proper edge-types and direction that connect the nodes of the corresponding matched-regions G_u^{mr} to the *first* p nodes in the pattern-region. The details of generating edge-bundles are

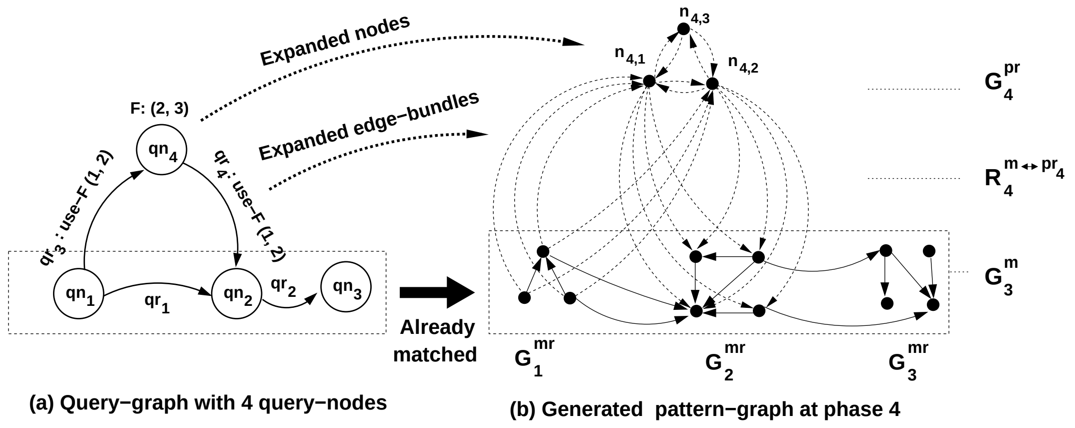


Figure 5.6: Generation of pattern graph G_4^p from the query-graph G^q at phase 4 using the imported/exported edge-bundles.

also discussed in Appendix B.

The rationale for such a query-edge expansion is to consider "*all possible situations where a subset of the source nodes can connect to a subset of the sink nodes which can be used in the context of graph pattern matching process*". The matching process then computes a group of edges that satisfy in a suboptimal way the min/max cardinality requirement for edge quantities among the group of nodes, according to the AQL query.

The pattern-graph G_l^p at phase $l = |N^q|$ (where $|N^q|$ is the number of query-graph nodes) is referred to as the *complete pattern-graph* $G^p = (N^p, R^p)$, where all query-nodes and query-edges have been expanded and the full pattern-graph has been generated. However, the incremental nature of the process allows to add more query-nodes and query-edges to the query-graph and extend the matching process by providing a more detailed AQL query.

5.2.7 Input-graph

The *input-graph* at phase i is an ARG $G_i^I = (N_i^I, R_i^I)$ which is a subgraph of the source-graph $G^s = (N^s, R^s)$. The input-graph G_i^I is represented using the graph summation, as:

$$G_i^I = G_{i-1}^m \; + \; (\mathcal{R}_i^{m \leftrightarrow sr_i} \; \oplus \; G_{g(i)}^{sr})$$

where G_{i-1}^m is the matched-graph at previous phase and $G_0^m = \phi$; $G_{g(i)}^{sr}$ is the "selected source-region" for the current phase i; and the connector-edges $\mathcal{R}_i^{m \leftrightarrow sr}$ are edges from the source-graph G^s between two subgraphs G_{i-1}^m and $G_{g(i)}^{sr}$. The input-graph G_i^I and pattern-graph G_i^p are supplied to the matching process at phase i to be matched and produce the matched-graph G_i^m.

5.3 Summary

In this chapter, the steps for generating a pattern-graph from an AQL query text that models the conceptual architecture of a system as an abstract module interconnection pattern, and the necessary graphs that model the matching process were discussed. Specifically, the macroscopic view of the pattern-graph consists of a number of smaller patterns to be matched with the graphs of the system modules, and groups of interconnection links to match with the import/export links among the modules. During the matching process the pattern-graph is incrementally matched at successive graph matching phases which simplifies the representation of the pattern-graph as a pattern-region that is connected to a collection of already matched graphs and their import/export links. The use of graph summation notations and the modular structure of the AQL query pattern allows to represent the architectural recovery as an incremental process that is modeled in terms of a recursive graph algebraic equation.

Chapter 6

Graph pattern matching

In most current approaches, software architecture recovery is viewed either as: a pattern matching problem [56, 79, 55, 48, 39]; a constraint satisfaction problem [113, 92]; a clustering problem [91, 78, 59, 107]; a composition and visualization problem [78, 38]; or a lattice partitioning problem [98, 68, 109]. The reverse engineering community has paid particular attention to the pattern matching approaches since they allow the use of domain knowledge and system constrains to be considered, and they can provide a user-assisted environment for architectural recovery. These pattern-matching approaches incorporate various strategies to identify the existence of a pattern in an information base that models the actual system being analyzed.

The most commonly used models for system representation are various forms of entity relationship graphs that are also currently used as means to represent high-level abstraction of a software system and serve as data exchange formats among various reverse-engineering tools [19, 49]. Motivated by the expressiveness and simplicity of entity relationship graphs this thesis proposes a model whereby architecture recovery is considered as an approximate graph matching problem between a graph that serves as an hypothesis (pattern) for the architecture of the system, and a graph that denotes the source-code of the system being analyzed (input). To limit the computational complexity of the matching process, an incremental algorithm is applied on specific regions of the software system entity relationship graph that represent different search domains.

In a nutshell, *graph matching* refers to algorithms for comparing two graphs G_1 and G_2 [95, 25, 108, 75] by the means of a function f that maps the nodes and edges of G_1 onto the nodes and edges of G_2. In the proposed software architecture recovery the process of generating the pattern-graph is exploratory in nature and the pattern-graph is not a fix graph. Therefore, finding the exact match between the pattern-graph and a subgraph of the software system graph is not feasible. On the other hand, in approximate graph matching the goal is to identify a subgraph of the input-graph that is similar to a given pattern-graph within a certain threshold value that will be further investigated by the user. In this context, an approximate graph matching algorithm can identify an optimal sequence of *graph edit operations* (i.e., node or edge insertion / deletion / relabeling) that are applied on one graph in order to make the two graphs isomorphic [95, 34, 25].

The contributions of this Chapter focus on: i) modeling the software architecture recovery as graph pattern matching problem; ii) proposing a multi-phase search algorithm to perform incremental matching of a modular pattern-graph at file-level or function-level; iii) proposing new cost functions for inter/intra-module node/edge insertion/deletion that incorporates hard constrains and similarity measure; and

iv) proposing a bounded-queue heuristic that relaxes the exponential running time complexity of the A^* algorithm.

This Chapter is structured as follows. First, an overview of the graph matching process is presented. Second, the graph pattern matching model of the architecture recovery is discussed. Third, a new graph distance based on different cost evaluations is provided. Fourth, the proposed bounded path-queue A^* search algorithm is presented. Finally, an example of a two-phase graph matching process is discussed.

6.1 Overview of the graph matching process

In this Section, an overview of the graph matching process is discussed. A formal representation of the matching process will be presented later in this Chapter.

6.1.1 Step 1: System representation

The software system is parsed and the source-code entities and data/control dependencies are abstracted according to an architectural-level domain model which yields the entity-relationship source-graph G^s. The source-graph provides 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. To address this problem, the search space is decomposed using data mining techniques to generate a collection of sub-spaces, each providing a subset of the initial search-space. Each sub-space corresponds to a source-region G^{sr}_j which is distinguished by the main-seed node n_j.

Each node in a source-region is labeled with a similarity (or closeness) value to the main-seed of the source-region as a means for the matching process to operate on groups of highly associated entities. The high-level representation of a software system is defined as:

$$system = (G^s,\ D(N^s))$$

where, $G^s = (N^s, R^s)$ and $D(N^s) = [D^{n_j} \mid j \in [1\ ..\ |N^s|]\ ]$ is the collection of domains of system entities in N^s. Each domain D^{n_j} consists of the source-region nodes along with the corresponding similarity values to the main-seed of the given source-region. 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.

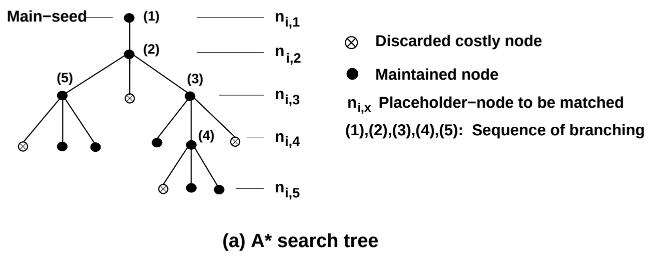
6.1.2 Step 2: pattern representation

An abstract pattern of *modules-and-interconnections* for the software system is modeled as a query in the proposed Architecture Query Language (AQL). An AQL query can be further represented as a query-graph consisting of composite nodes that are linked through composite edges as discussed earlier. The composite nodes and composite edges of the query-graph (i.e., pattern) are expanded into pattern-regions and edge-bundles that are consequently matched against the source-regions and their connector-edges. The rationale for expanding the composite-edges is to allow every subset of the nodes in a source module to be connected to every subset of the nodes in the destination module, according to the constraints modeled in the AQL query.

6.1.3 Step 3: graph matching process

The matching process computes a sub-optimal match between a pattern-graph that originates from an AQL query and an input-graph that originates from the system source-graph. The matching process is performed in k phases (k is the number of AQL query modules) with the requirement that the obtained results conform with the constraints specified by the AQL query. The current matching phase is denoted as "i".

We use the A^* search algorithm modified by a "bounded-queue heuristic" to compute a sub-optimal matching cost between the pattern-graph and input-graph while the AQL query constraints are not violated. The search algorithm generates a search-tree that corresponds to the recovery of each module M_i in AQL query (Figure 6.1(a)), that consists of: i) a *root node* for matching the main-seed n_j of the source-region G_j^{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 edit operations 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. 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 AQL module Mi).



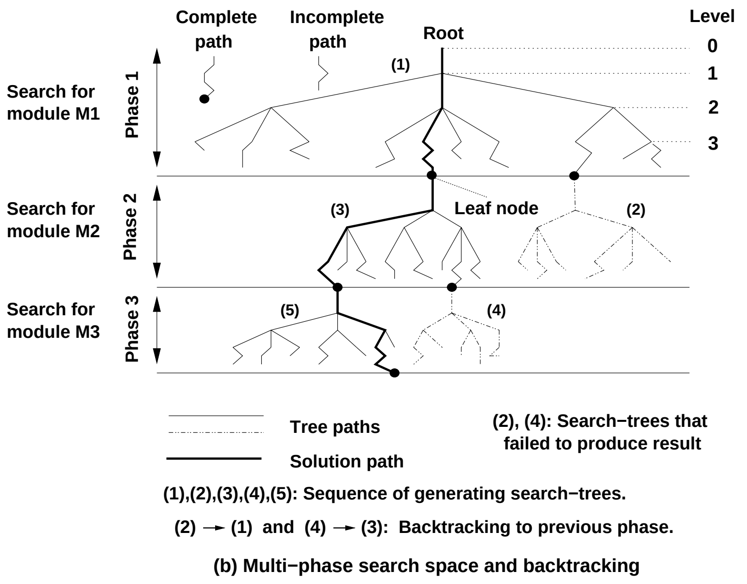

Complete path
Incomplete path
Root
Level
0
1
2
3
(1)
Search for module M1
Phase 1
Search for module M2
Phase 2
Leaf node
(3)
(2)
Search for module M3
Phase 3
(5)
(4)
Tree paths
Solution path
(2), (4): Search−trees that failed to produce result
(1),(2),(3),(4),(5): Sequence of generating search−trees.
(2) → (1) and (4) → (3): Backtracking to previous phase.
(b) Multi−phase search space and backtracking

Figure 6.1: Demonstration of a multi-phase search strategy using: (a) an A^* optimal search algorithm to match the placeholder-nodes at each phase and; (b) backtracking between phases.

Multi-phase matching and backtracking

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 $i \in [1..|N^q|]$, where $|N^q|$ is the number of nodes in the query-graph G^q). 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. In this form, the whole matching process is divided into k incremental phases (as k partial-matchings) so that the recovery process performs a *multi-phase* matching. Each partial-matching at phase i (i : $1, 2, 3, ..., k$) generates a search-tree which is a part of the multi-phase search-space, illustrated in Figure 6.1(b).

If the current phase i of the matching process fails to identify a matched-graph G_i^m (i.e., AQL link constraints can not be satisfied) then the control algorithm *backtracks* by: i) discarding the result of the previous phase, G_{i-1}^m; ii) restoring the search-tree for previous phase $i-1$; iii) expanding the search-tree to find another solution $G_{i-1}^{m'}$, and; iv) advancing to the current phase i and generating a new search-tree from $G_{i-1}^{m'}$. In Figure 6.1(b), each search-tree with dashed lines represent an unsuccessful search that caused the search mechanism to backtrack to the previous search-space, generate another leaf-node, and then create a new search-tree (shown as solid-lines). A thick line from the root of the first tree to a leaf of the last tree represents a path that yields a sub-optimal solution, since the number of alternative paths is bounded by the "bounded queue" heuristic.

6.2 Software architecture recovery as graph pattern matching

In this Section, "software architecture recovery" is defined as a "graph pattern matching problem", and is modeled in the form of recursive graph equations that correspond to an iterative graph matching process. In defining the problem, the notion of *graph distance* is used as follows:

Graph distance: the *distance* between two attributed relational graphs $G1$ and $G2$, denoted as $dist(G1, G2)$, is defined as the minimum cost of a sequence of graph edit operations that must be performed on one graph (e.g., $G1$) so that it becomes isomorphic to the other graph (i.e., $G2$). These changes are usually in the form of node or edge *deletion*, *insertion*, or *relabeling* [34].

Software Architecture Recovery (A, G^s, d_t)

Given an AQL query A that specifies an interaction pattern between system modules (or subsystems) and is represented as a pattern-graph G^p, a source-graph G^s that represents the data/control dependencies in a software system, and a graph distance threshold d_t that corresponds to the interaction constraints between the modules, the problem is to find a subgraph of G^s (namely G^m) among the different alternative subgraphs whose graph distance with respect to the pattern-graph G^p is both minimum and less than the distance threshold d_t, that is:

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

In this problem definition, the distance threshold d_t ensures that: i) the hard constraints imposed by the module-interconnection pattern of the AQL query A with respect to the minimum/maximum number of import/export relations will be satisfied; and ii) the minimum required quality of the recovered modules with respect to the average closeness value of each module will be satisfied, otherwise no result can be produced. A search algorithm such as the proposed bounded-queue A^* is applied to find a sub-optimal graph-distance between the pattern-graph and all the alternative subgraphs of the source-graph G^s.

The details of measuring the distance between two graphs and the search algorithm to implement the matching process are discussed in the following Sections.

6.3 Modeling the graph matching process

The proposed multi-phase, incremental, and approximate graph matching process can be modeled in the form of the recursive equations illustrated in Figure 6.2.

In this equational form, the whole graph-matching process is performed in $|N^q|$ iterations (phases), where $|N^q|$ is the number of nodes in the query-graph G^q. At each phase i ($i \in [1..|N^q|]$) the result of matching at previous phase G^m_{i-1} is used to build an input-graph G^I_i and a pattern-graph G^p_i to be matched and produce a matched-graph G^m_i, which in turn is used to build G^I_{i+1} and G^p_{i+1} for the next matching phase $i+1$, and so on. In this context, G^m_0 is defined as a Nil graph with zero number of nodes and edges, and when $i = |N^q|$ then $G^I_i = G^I, G^p_i = G^p$, $G^m_i = G^m$, and the matching process terminates.

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. $G_i^m = G_{i-1}^m \; + \; (\mathcal{R}_i^{m \leftrightarrow mr_i} \; \oplus \; G_i^{mr})$
6. $G_i^m = \sum_{u=1}^{i} \; (G_u^{mr} \; \oplus \; \mathcal{R}_u^{mr^* \leftrightarrow mr_u})^1$
7. $G_i^I = G^I, \quad G_i^p = G^p, \quad G_i^m = G^m \qquad if \;\; i = |N^q|$

Figure 6.2: The recursive equations for the proposed multi-phase, incremental, and approximate graph matching process.

In the above equations, the input-graph G_i^I (equation 2) consists of the matched-graph at previous phase G_{i-1}^m, the selected source-region $G_{g(i)}^{sr}$, and the source-graph connector-edges $\mathcal{R}_i^{m \leftrightarrow sr_i}$. The pattern-graph G_i^p (equation 3) consists of the matched-graph at previous phase G_{i-1}^m, the pattern-region G_i^{pr}, and the edge-bundles $\mathcal{R}_i^{m \leftrightarrow pr_i}$. 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 the pattern-graph G_i^p. A subgraph G_i^x with minimum graph distance to the pattern-graph is the solution of the matching process, i.e., G_i^m. The resulting matched-graph G_i^m (equation 5) consists of matched-graph at previous phase G_{i-1}^m, matched-region G_i^{mr}, and connector-

1
$$G_i^m = (G_1^{mr} \oplus \mathcal{R}_1^{mr^* \leftrightarrow mr_1}) \; + \; .... \; + \; (G_i^{mr} \oplus \mathcal{R}_i^{mr^* \leftrightarrow mr_i})$$

edges $\mathcal{R}_i^{m\leftrightarrow mr_i}$. The modular view of the resulting matched-graph G_i^m (equation 6) consists of a collection of matched-regions G_u^{mr} (i.e., recovered modules) and their connector-edges (i.e., import/export links) to other matched-regions.

Figure 6.3(a) illustrates a model of the problem and solution for software architecture recovery in the form of graph equations. Figure 6.3(b) illustrates an example of the matching process at phase 2 that will be further discussed in Section 6.6.

6.4 Graph distance

This Section outlines the concepts pertinent to measuring the distance between the pattern-graph G_i^p and a subgraph of the input-graph G_i^I so that a sub-optimal match can be computed. The measured graph distance is used by the graph-matching algorithm to select a sub-optimal minimum distance subgraph of G_i^I as the matched-graph G_i^m. The graph distance is related to the cost of graph edit operations on the "pattern-region G_i^{pr} and its connector-edges $\mathcal{R}_i^{m\leftrightarrow pr_i}$" to match them against a subgraph of the "selected region $G_{g(i)}^{sr}$ and the connector-edges $\mathcal{R}_i^{m\leftrightarrow sr_i}$". These edit operations result in a transformation or instantiation of G_i^{pr} that yields a "matched-region G_i^{mr} and the connector-edges $\mathcal{R}_i^{m\leftrightarrow mr_i}$" (Figure 6.3(b)).

A certain cost is associated with each graph edit operation that corresponds to matching a node n_k from the selected source-region $G_{g(i)}^{sr}$ with a placeholder-node $n_{i,j}$ from the pattern-region G_i^{pr}. The costs for edge-insertion, edge-deletion, node-insertion, and node-deletion, denoted as c^{ei}, c^{ed}, c^{ni}, and c^{nd} are discussed below. In this thesis, the following graph distance function is used, whose elements are

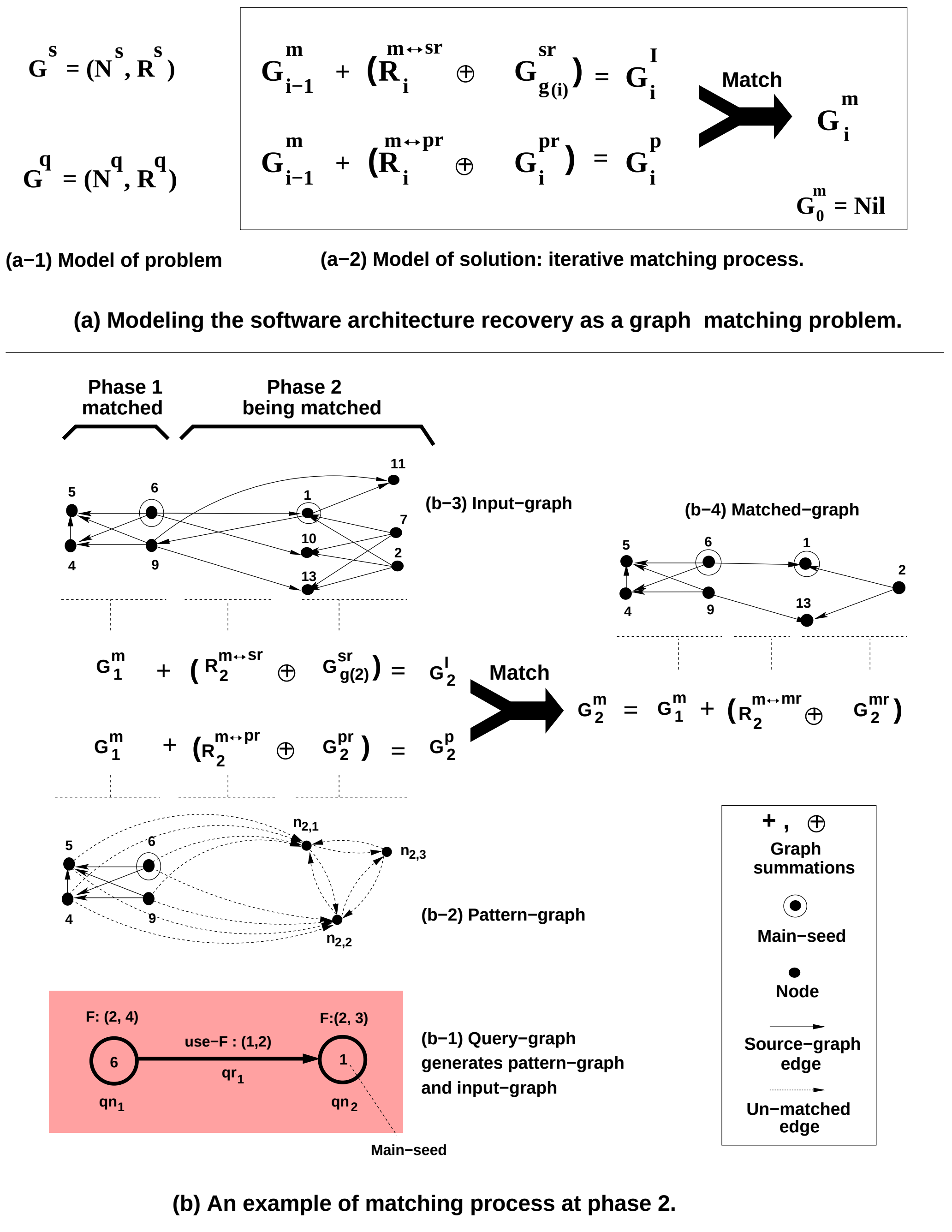

G^s = (N^s, R^s)
G^q = (N^q, R^q)
G^m_{i−1} + (R_i^{m↔sr} ⊕ G^{sr}_{g(i)}) = G^I_i
G^m_{i−1} + (R_i^{m↔pr} ⊕ G^{pr}_i) = G^p_i
Match
G^m_i
G^m_0 = Nil
(a−1) Model of problem
(a−2) Model of solution: iterative matching process.
(a) Modeling the software architecture recovery as a graph matching problem.
Phase 1 matched
Phase 2 being matched
(b−3) Input−graph
(b−4) Matched−graph
G^m_1 + (R_2^{m↔sr} ⊕ G^{sr}_{g(2)}) = G^I_2
G^m_1 + (R_2^{m↔pr} ⊕ G^{pr}_2) = G^p_2
Match
G^m_2 = G^m_1 + (R_2^{m↔mr} ⊕ G^{mr}_2)
n2,1
n2,3
n2,2
(b−2) Pattern−graph
F: (2, 4)
F:(2, 3)
use−F : (1,2)
qr1
qn1
qn2
Main−seed
(b−1) Query−graph generates pattern−graph and input−graph
+ , ⊕ Graph summations
Main−seed
Node
Source−graph edge
Un−matched edge
(b) An example of matching process at phase 2.

Figure 6.3: Modeling a software architecture recovery process as a graph-matching problem. The graph pattern-matching process iteratively matches a *pattern-graph* with an *input-graph* and yields a *matched-graph*. A *query-graph* provides all required information for different phases of an iterative matching process.

defined below:

$$dist(G_i^p, G_i^x) = \sum_{j=1}^{|N_i^{pr}|} (c_{in}^{ed} + c_{out}^{ed} + c_{out}^{ei} + c^{nd})|_{n_{i,j} \ matches \ with \ n_{k_j}}$$

where, $n_{k_j} \in N_i^x$ and $|N_i^x| \leq |N_i^{pr}|$ because of possible deletion of node $n_{i,j}$ from G_i^{pr}. Two subscripts "*in*" and "*out*" denote the costs that occur inside the pattern-region or on the connector-edges $\mathcal{R}_i^{m \leftrightarrow pr_i}$. The resulting costs are defined below.

6.4.1 Edge insertion cost

Two cases are defined as follows:

a) Inside-pattern edge insertion cost, c_{in}^{ei}

This cost is not applicable, since the pattern-region is already maximally expanded by its definition.

b) 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.

6.4.2 Edge deletion cost

Two cases are defined as follows:

a) Inside-pattern edge deletion cost, c^{ed}_{in}

The objective of defining the *inside-pattern edge deletion* cost c^{ed}_{in} is to be able to recover cohesive modules and subsystems as matched-regions G^{mr}_i during the matching process at phase i. In order to recover cohesive modules, a pattern-region G^{pr}_i is generated as a fully connected graph so that the matching process can reveal the most intra-related group of entities in the source-region $G^{sr}_{g(i)}$. The cost c^{ed}_{in} is defined as:

$$c^{ed}_{in} = c^{ed}_{in_1} + c^{ed}_{in_2} \qquad where$$

$$c^{ed}_{in_1} = \frac{M-s}{k} \qquad and \qquad c^{ed}_{in_2} = \frac{0.25d\ s}{k}$$

where, M is the maximum similarity value among all pairs of entities in the corresponding source-region $G^{sr}_{g(i)}$; s is the similarity value between two entities that the edge to be deleted is connected; d is the number of deleted edges; and k is the number of already matched nodes.

The sub-cost $c^{ed}_{in_1}$ is denoted as the *default dissimilarity* value between the candidate-node n_k and each matched node in G^{pr}_i; whereas, the sub-cost $c^{ed}_{in_2} = \frac{0.25d\ s}{k}$ 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 $\frac{0.25s}{k}$ to the default cost value $\frac{M-s}{k}$, and at worst case (two edge deletion) the dissimilarity value doubles. Therefore, increasing the coefficient 0.25 causes that the

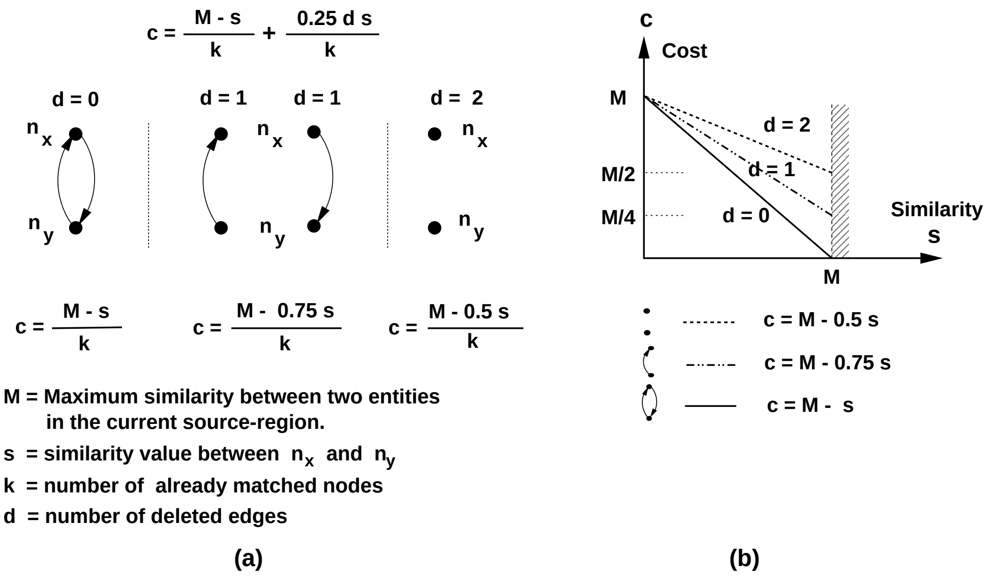

c = (M - s)/k + (0.25 d s)/k
d = 0
d = 1
d = 1
d = 2
n x
n y
c = (M - s)/k
c = (M - 0.75 s)/k
c = (M - 0.5 s)/k
M = Maximum similarity between two entities in the current source-region.
s = similarity value between n x and n y
k = number of already matched nodes
d = number of deleted edges
(a)
c
Cost
M
M/2
M/4
d = 2
d = 1
d = 0
Similarity
s
M
c = M - 0.5 s
c = M - 0.75 s
c = M - s
(b)

Figure 6.4: (a) Definition of inside-pattern edge deletion cost c_{in}^{ed}. (b) Diagram for cost versus similarity when only one node already matched in G_i^{pr}, i.e., k = 1.

missing edge to become more important than the dissimilarity value, and vice versa.

The costs for different cases of inside-pattern edge deletion are as follows:

- $c_{in}^{ed} = \frac{M - s}{k}$: zero edge deletion d = 0
- $c_{in}^{ed} = \frac{M - 0.75s}{k}$: one edge deletion d = 1
- $c_{in}^{ed} = \frac{M - 0.5s}{k}$: two edge deletion d = 2

Figure 6.4(a) illustrates different cases for inside-pattern edge deletion costs, and the diagram in Figure 6.4(b) demonstrates the variation of the cost versus similarity between two nodes with the number of deleted edges d as the parameter.

In defining cost c_{in}^{ed} the following properties are considered:

- The cost c_{in}^{ed} depends on both the similarity value $entAssoc(n_x, n_y)$ between

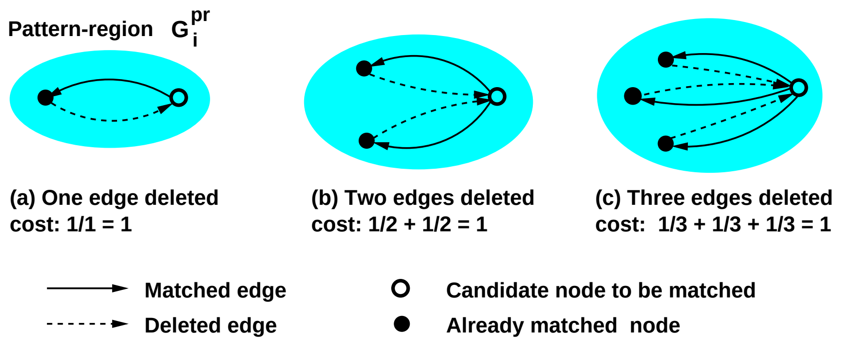


Figure 6.5: An average of the inside-pattern edge deletion cost compensates for the number of nodes in the pattern-region G_i^{pr}.

the candidate-node n_x and each of the already matched node n_y in G_i^{pr}, and the number of deleted edges between n_x and those nodes. Note that, the similarity metric $entAssoc(n_x, n_y)$ is independent of the edges between n_x and n_y. Specifically, in an associated group g_z, defined in Section 3.3, there may be no edges between two nodes in the itemset (shared nodes) or basketset (sharing nodes) but still these nodes are considered similar to each other.

- In the clustering literature, the cost measure (dissimilarity or distance) is commonly defined as the complement of the similarity measure [36] such as:

$$c = M - s$$

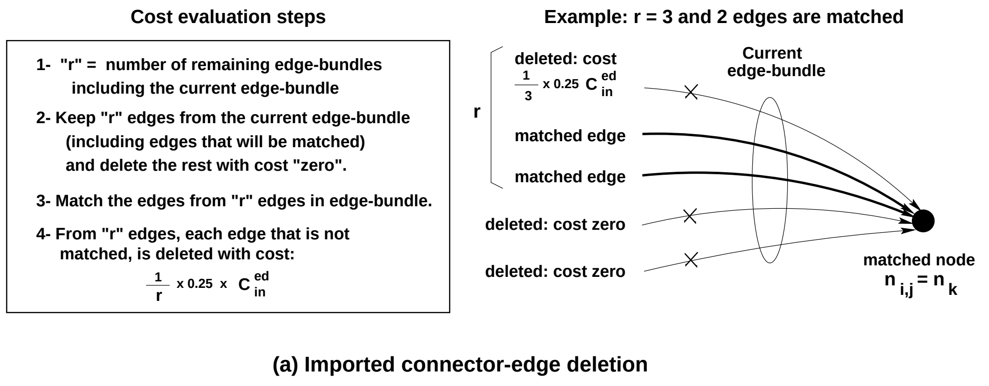
Where, c and s are the cost and similarity values between two entities, respectively; and M is the maximum similarity value among all pairs of entities in the corresponding source-region (i.e., $M = 1$, if the similarity metric is normalized).

- The cost of matching is proportional to the number of already matched nodes in pattern-region G_i^{pr}. Figure 6.5 illustrates three cases with different number of nodes already matched in G_i^{pr} which can result in different cost values. However, in all cases the candidate-node has one edge to each node in graph G_i^{pr} which implies a similar relation to the group of matched nodes. In order to compensate for the number of matched nodes in G_i^{pr}, the cost for each edge deletion is divided by the number of matched nodes in G_i^{pr} which yields the same average value for all three cases.

b) Connector-edge deletion cost, c_{out}^{ed}

The cost in this Section is calculated 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. 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. In Appendix B, the procedure for generating edge-bundles $\mathcal{R}_i^{m \leftrightarrow pr_i}$ for pattern-region G_i^{pr} indicates that for each connector-edge that is to be matched, one edge-bundle is generated. As an example, for a query-edge qr_k with minimum cardinality 1 and maximal cardinality 3, three edge-bundles are initially generated. Therefore, during the matching process, for each connector-edge that is matched, one edge-bundle must be deleted. The number of remaining edge-bundles indicates the number of connector-edges that can still be matched 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*" as follows:


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:
$\frac{1}{r} \times 0.25 \times C^{ed}_{in}$
Example: r = 3 and 2 edges are matched
r
deleted: cost $\frac{1}{3} \times 0.25\ C^{ed}_{in}$
matched edge
matched edge
deleted: cost zero
deleted: cost zero
Current edge-bundle
matched node
$n_{i,j} = n_k$
(a) Imported connector-edge deletion

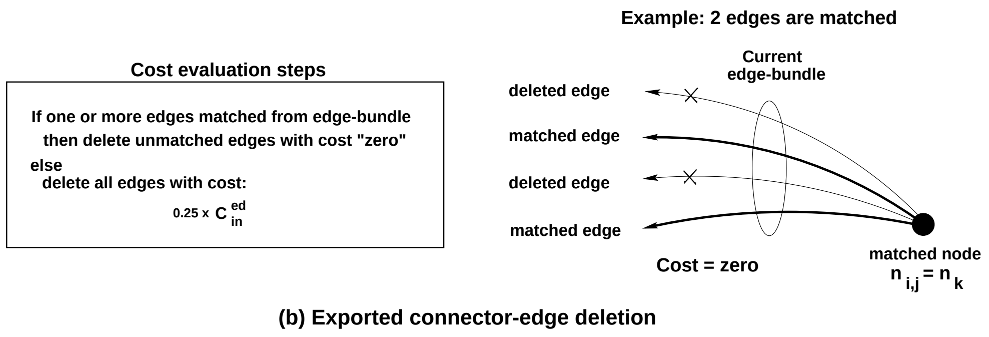

Example: 2 edges are matched
Cost evaluation steps
If one or more edges matched from edge-bundle then delete unmatched edges with cost "zero"
else
delete all edges with cost:
$0.25 \times C^{ed}_{in}$
deleted edge
matched edge
deleted edge
matched edge
Current edge-bundle
Cost = zero
matched node
$n_{i,j} = n_k$
(b) Exported connector-edge deletion

Figure 6.6: The cost evaluation for deleting import/export connector-edges corresponding to the pattern-region.

- **b-1) Imported connector-edge deletion:** as discussed above, for matching each imported connector-edge a whole imported edge-bundle must be deleted with no-cost. However, within the edge-bundle that is connected to the current place-holder node $n_{i,j}$, for each edge deletion a cost is applied. The cost evaluation steps along with an example are illustrated in Figure 6.6(a). In this cost, "r" is the number of remaining edge-bundles including the current edge-bundle, and 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

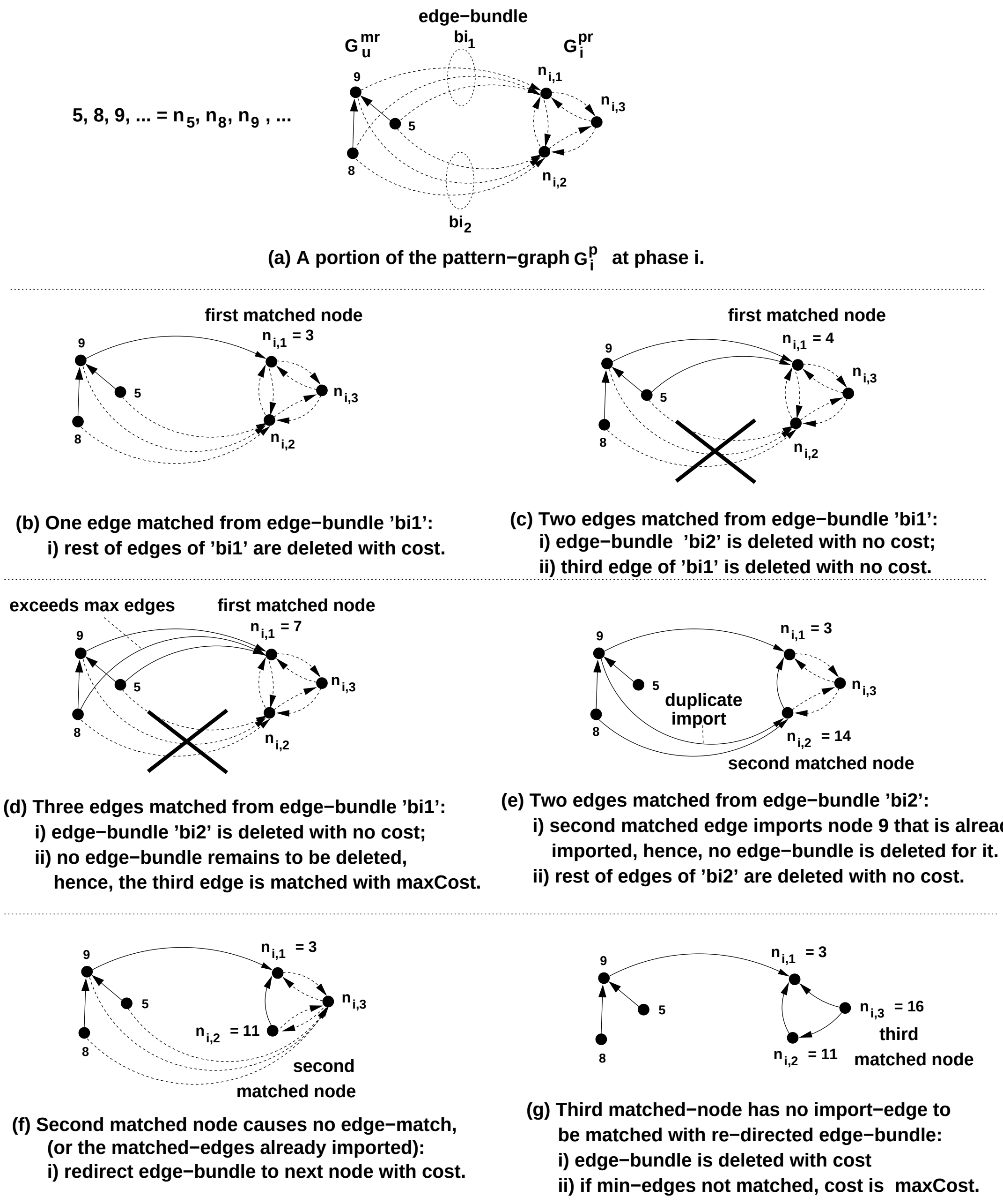
in augmenting the number of matched imported edges to reach to its maximum number. Therefore, matching more imported edges by the current node means less cost. The cost is calculated based on the eligibility of the node to produce a cohesive module, by taking into account the *inside-pattern edge deletion 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 imported connector-edges. The edge deletion cost is as follows:

$$c_{out}^{ed} = \begin{cases} 0 & \text{for extra edges that are more than "}r\text{" edge} \\ \frac{1}{r} \times \ 0.25 \times c_{in}^{ed} & \text{for edges that are not matched within "}r\text{" edges} \end{cases}$$

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.

Example: cost evaluation for imported connector-edge deletion

Figure 6.7 illustrates several cases of matching imported connector-edges where the maximum cardinality is 2. The following explanations are with reference to the parts of Figure 6.7.



(a) A portion of the pattern−graph G_i^p at phase i.

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

(c) 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.

(d) 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.

(e) 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.

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

(g) 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 6.7: Examples of edge deletion from *imported edge-bundles* $\mathcal{R}_i^{mr_u \to pr_i}$ that connect an already matched-region G_u^{mr} (a link-module) to the pattern-region G_i^{pr}.

Part (a): a portion of pattern-graph G_i^p with two imported edge-bundles bi_1 and bi_2 from an already matched-region G_u^{mr} (a link-module) to the pattern-region G_i^{pr} is shown, where the collection bi_1 and bi_2 constitute $\mathcal{R}_i^{mr_u \rightarrow pr_i}$, and $r = 2$.

Part (b): from the first edge-bundle one edge is matched, one edge is deleted with cost zero, and one edge is deleted with cost $\frac{1}{2} \times 0.25 \times c_{in}^{ed}$. One more edge can still be matched since bi_2 has not been considered yet.

Part (c): from the first edge-bundle two edges are matched and the third edge is deleted with cost zero. The whole second edge-bundle is deleted with cost zero, and no other edges can be matched since the maximum number of imported edges has been reached.

Part (d): from the first edge-bundle all three edges are matched; the second edge-bundle is deleted with cost zero. Since the number of the matched edges exceeds the maximum number of edges (r = 2), the third edge is matched with *maxCost* (see cost assignment for c_{out}^{ei}).

Part (e): from the second edge-bundle two edges are matched; the second matched edge imports a duplicate node, hence it is not considered as a matched edge and only one edge is considered as matched[2]. The third edge of the edge-bundle is deleted with cost zero. This case can occur after the case in part (b).

Part (f): from the second edge-bundle no edges are matched and the remaining edge-bundles $r = 1$: cost for not matching one edge is applied which is

[2]This restriction is in conformance with semantics of IMPORTS part of a COMPONENT in the AQL query Section 4.2.4.

$\frac{1}{1} \times 0.25 \times c_{in}^{ed}$, and the edge-bundle is redirected to the third placeholder-node. This case can occur after part (b)

Part (g): this case can occur after part (f) where the remaining edge-bundles is $r = 1$ and from the second edge-bundle no edges are matched. In this case, the edge-bundle can not be redirected since no placeholder-node is left, therefore all edges of edge-bundle are deleted with cost for not matching one connector-edge, i.e., $\frac{1}{1} \times 0.25 \times c_{in}^{ed}$. However, if the minimum number of the connector-edges that is specified in the AQL query is 2, then the cost is *maxCost* since only one connector-edge has been imported to the current matched-region G_i^{mr} which violates the minimum threshold.

- **b-2) Exported connector-edge deletion:** the cost of deleting edges from the current edge-bundle is as follows: if one or more edges are matched then the rest of unmatched edges are deleted with cost zero; otherwise the edge-bundle is redirected or deleted with cost:[3]

$$c_{out}^{ed} = 0.25 \times c_{in}^{ed}$$

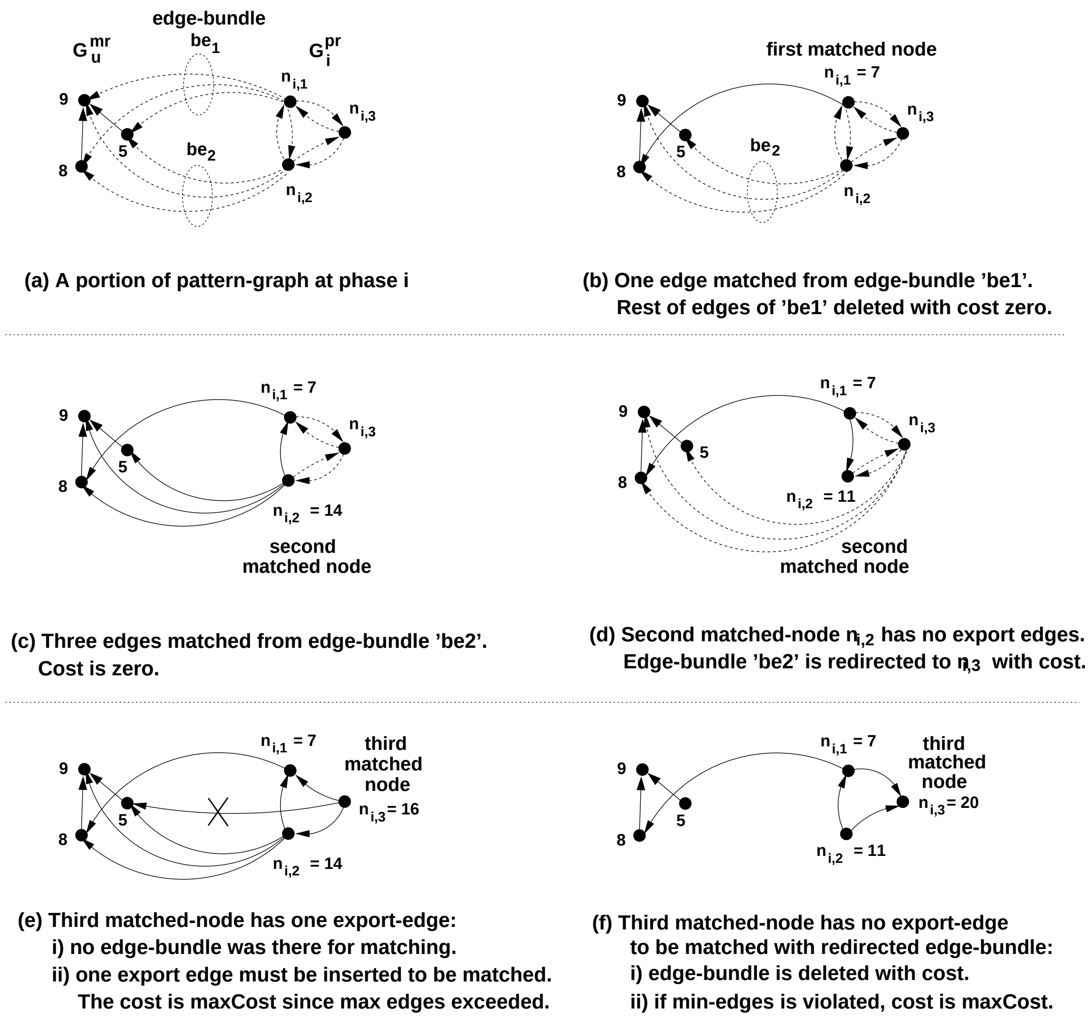
However, this cost evaluation is valid only if the minimum (maximum) number of the exported edges from the pattern-region G_i^{pr} still can be reached (not exceeded) by the current node-matching. Otherwise, the cost is *max-*

[3]For any number of exported connector-edges that are matched during the current node-matching $n_{i,j}$ with n_k, only one exported edge-bundle (i.e., current edge-bundle) is affected and a part of its edges can be deleted. The reason is that no matter how many exported edges exist between the matched node n_k and the nodes in linked-module G_u^{mr}, still only one node n_k is exported, and hence one edge-bundle must be affected.

Cost and the current node-matching will be discarded. Figure 6.6 illustrates the corresponding cost assignment. The justification of this cost assignment is similar to that of imported connector-edge deletion.

Example: cost evaluation for exported connector-edge deletion
Figure 6.8 illustrates several cases of matching exported connector-edges where the maximum cardinality is 2. The following explanations are with reference to the parts of Figure 6.8.

Part (a): a portion of the pattern-graph G_i^p with two exported edge-bundles be_1 and be_2 from the pattern-region G_i^{pr} to a matched-region G_u^{mr} (link-module) is shown, where the collection be_1 and be_2 constitute $\mathcal{R}_i^{mr_u \leftarrow pr_i}$.
Part (b): from the first edge-bundle one connector-edge is matched and the rest of edges are deleted with cost zero.
Part (c): from the second edge-bundle three edges are matched, no edges are deleted, and the cost is zero. This case can occur after part (b).
Part (d): from the second edge-bundle no edges are matched, hence the cost of not matching one edge is applied which is $0.25 \times c_{in}^{ed}$. The edge-bundle is redirected to the third placeholder-node $n_{i,3}$.
Part (e): from the third node-matching one edge is matched: no edge-bundle is there for edge-matching; one connector-edge must be inserted for edge-matching with cost *maxCost* (see cost assignment for c_{out}^{ei}). The reason is that the maximum number of connector-edges is exceeded. This case can



(a) A portion of pattern-graph at phase i

(b) One edge matched from edge-bundle 'be1'.
Rest of edges of 'be1' deleted with cost zero.

(c) Three edges matched from edge-bundle 'be2'.
Cost is zero.

(d) Second matched-node $n_{i,2}$ has no export edges.
Edge-bundle 'be2' is redirected to $n_{i,3}$ with cost.

(e) Third matched-node has one export-edge:
i) no edge-bundle was there for matching.
ii) one export edge must be inserted to be matched.
The cost is maxCost since max edges exceeded.

(f) Third matched-node has no export-edge
to be matched with redirected edge-bundle:
i) edge-bundle is deleted with cost.
ii) if min-edges is violated, cost is maxCost.

Figure 6.8: Examples of edge deletion from *exported edge-bundles* $\mathcal{R}_i^{mr_u \leftarrow pr_i}$ that connect the pattern-region G_i^{pr} to an already matched-region G_u^{mr} (link-module).

occur after part (c).

Part (f): this case can occur after part (d) where the second edge-bundle was redirected to the third placeholder-node. In this case no edge is matched from the third node-matching and the edge-bundle is deleted with cost $0.25 \times c^{ed}_{in}$. However, if the minimum number of connector-edges that is specified in the AQL query is 2 (i.e., the originally expanded query-edge is $qr_k(2,2)$), then the cost is *maxCost* since only one node is exported.

6.4.3 Node insertion/deletion cost

Two cases are defined as follows:

a) Node insertion cost, c^{ni}

This cost is not applicable, since the nodes of pattern-region G^{pr}_i are already maximally expanded by definition, therefore, any node insertion inside G^{pr}_i violates the maximum number of nodes defined in the AQL query.

b) Node deletion cost, c^{nd}

In this cost, the number of nodes in the result of matching at phase i (i.e., matched-region G^{mr}_i) must be within a size range (*min, max*) associated with the query-node qn_i. Therefore, some of the nodes in G^{pr}_i can be deleted and still produce a valid result. This situation occurs when there is no combination of p ($p = |N^{pr}_i|$) nodes from the selected source-region $G^{sr}_{g(i)}$ that can produce a matching result. In this case, during the matching process the algorithm reduces the number of nodes in the pattern-region G^{pr}_i to produce a result. This is performed by matching some of the nodes from G_i^{pr} with the *null* node Λ, and replacing their connected edges by *null edges*, either (Λ, n_x) or (n_x, Λ). During the matching process, if the number of null nodes in the pattern-region G_i^{pr} is less than the value "$max - min$" then the cost of deleting the current node is the cost c_{in}^{ed} of matching an example node n_x (with minimum similarity value s_{min} in the source-region $G_{g(i)}^{sr}$) where no edges exist between n_x and the already matched nodes in the pattern-region G_i^{pr}, i.e.:

$$c^{nd} = M - 0.5 \times s_{min}$$

where M and s_{min} are the maximum and minimum similarity values between two entities in the current source-region $G_{g(i)}^{sr}$. If the number of null nodes in the pattern-region is more than "$max - min$" then the cost c^{nd} is *maxCost*.

6.5 Bounded queue A^* search algorithm (BQ-A^*)

In the A^* algorithm [86] a search-tree with *incomplete tree-paths* is built, as illustrated in Figure 6.9(a). The sequence of expanded tree-nodes from the root of the search-tree to any other tree-node is called a *tree-path*. There are two costs associated with a tree-node tn in a search-tree which are illustrated in Figure 6.9(b), as: i) *path-cost* c_{path}, which is the cost of a tree-path from the root to the tree-node tn; and ii) *underestimated-cost* c_{uest}, which is the estimated cost of the cheapest path from tn to the goal tree-node. The function that computes such a cost is known as *heuristic function* [86]. The estimated cost of the cheapest solution through tree-node tn, denoted as c_{total}, is defined using these two costs. At each step the al-

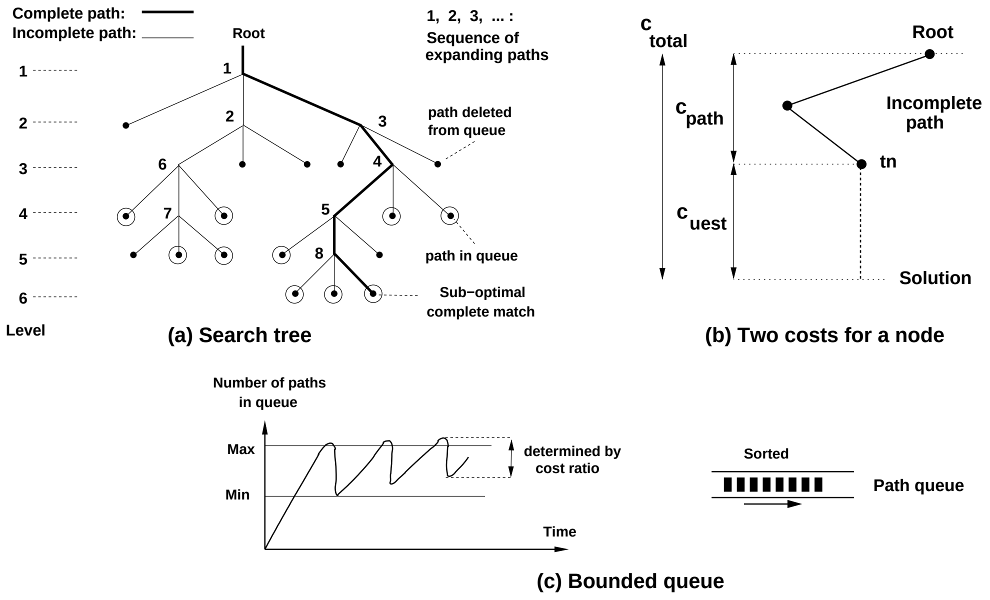


Figure 6.9: (a) Bounded queue A^* search tree with existing and deleted paths. (b) The costs that are associated with a search-tree node. (c) Bounded queue, where the number of paths in the queue is bounded between two thresholds.

gorithm expands an incomplete tree-path with the lowest cost for $c_{total} = c_{path} + c_{uest}$ among all other incomplete paths. Upon expansion, new incomplete paths are generated and added to the previous paths. The procedure continues until a *complete tree-path* which is an optimal solution is found. A valuation function allocates a cost to each node of the A^* search tree to guide the search process according to the costs that were defined in the Section 6.4.

Bounded queue

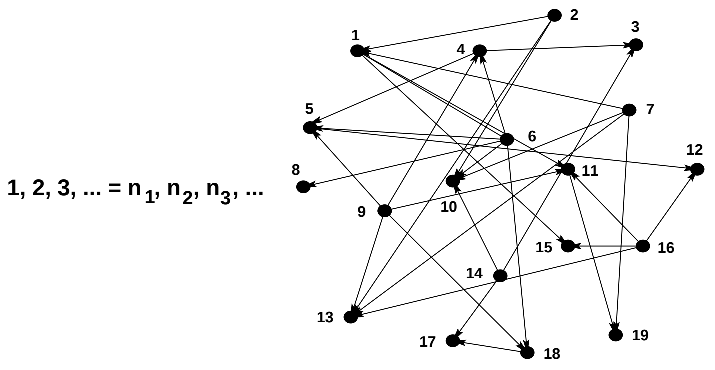
A major drawback of the optimal search algorithms is the requirement to maintain all incomplete tree-paths (partially-matched graphs) in a sorted queue that allows to select the cheapest tree-path to expand next. This queue grows very fast and in the worst case can have an exponential size, which makes 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. Figure 6.9(c) illustrates the oscillation of the number of paths in the path queue. Once the size of queue passes a maximum threshold, it is truncated to the minimum size. However, we only delete the paths from the tail of the path queue whose costs are much higher than those on the head of the queue. Therefore, when we collect paths whose costs are close to each other, the size of queue is kept around the maximum size, as shown in Figure 6.9(c). As a result the bounded queue heuristic yields a sub-optimal version of the A^* search algorithm as a trade for increasing the performance[4]. We denote the "bounded queue A^*" algorithm as: *BQ-A^** algorithm through out this thesis. In Figure 6.9(a), an example of a sub-optimal search with the sequence of path expansion is shown. In this simplified example, each incomplete path will expand to three paths and the (max, min) threshold is (16, 10).

[4]In practice, for a medium size system (#50 KLOC) we use a (max, min) queue size threshold of (400, 200) with ratio $\frac{maxScore}{minScore} = 2.0$.

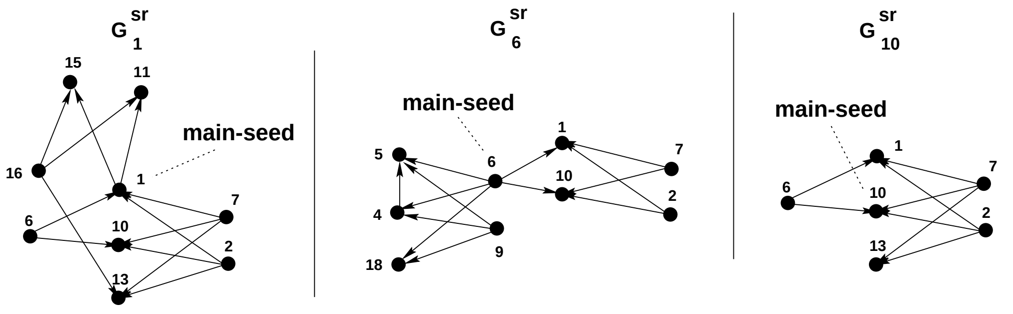
6.6 Example of the matching process

In this Section, an example of a two-phase graph pattern-matching process is discussed. In Figure 6.10, the graph matching is shown as the problem of finding a sub-optimal match between a subgraph of the source-graph G^s, against a pattern represented as a query-graph G^q with two composite-nodes and one composite-edge. The collection of source-regions in Figure 6.10(b) are represented as the *region representation* of the source-graph, i.e., $(G^s, D(N^s))$. The incremental matching is performed in two phases. In the first phase, the matched-graph $G_1^m = match(G_1^p, G_1^I)$ is computed, where the pattern-graph is $G_1^p = G_1^{pr}$ and the input-graph is $G_1^I = G_{g(1)}^{sr} = G_6^{sr}$ (Figure 6.11). In the second phase, the matched-graph $G_2^m = match(G_2^p, G_2^I)$ is computed, where the pattern-graph $G_2^p = G_1^m + (\mathcal{R}_2^{m \to pr_2} \oplus G_2^{pr})$ and the input-graph $G_2^I = G_1^m + (\mathcal{R}_2^{m \to sr} \oplus G_{g(2)}^{sr})$, and $G_{g(2)}^{sr} = G_1^{sr}$ (Figure 6.13).

The selected source-regions to be matched against the pattern-regions are indicated in the query-graph through the main-seed of each query-node, Figure 6.10(c). An algorithm for selecting a source-region (or equivalently a main-seed) is discussed in detail in [90]. The algorithm provides a ranked list of source-regions for the user to select from. The best source-region to select should be large, contain highly related entities, and be sufficiently distinct from the previously recovered modules. In this simple example, the source-regions G_6^{sr} (i.e., $G_{g(1)}^{sr}$) is the best among the three regions and hence, is selected for matching with G_1^{pr} at phase 1. G_6^{sr} has the highest average similarity value among the regions and has a high number of nodes. Next, the source-region G_1^{sr} (i.e., $G_{g(2)}^{sr}$) is selected to be matched against



(a) A source-graph $G^S = (N^S, R^S)$



Domain of main-seed 1

node	7	10	2	13	6	11	16	15
sim	4	4	4	4	3.5	3	3	3

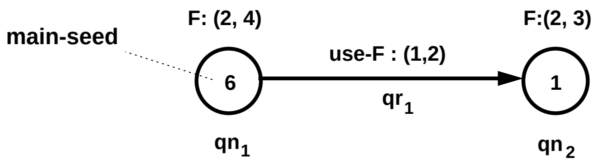
Domain of main-seed 6

node	5	9	4	18	1	7	10	2
sim	4	4	4	4	3.5	3.5	3.5	3.5

Domain of main-seed 10

node	1	7	13	2	6
sim	4	4	4	4	3.5

(b) Three source-regions of source-graph G^S representing $(G^S, D(N^S))$



(c) A query-graph with two composite-nodes and one composite-edge.

Figure 6.10: Illustration of a two-phase graph pattern-matching problem. (a) A raw source-graph $G^s = (N^s, R^s)$ as the whole search-space. (b) The source-graph G^s is decomposed into a collection of source-regions that are represented as domains of nodes $D(N^s)$ and the set of edges R^s from G^s, that collectively represent $(G^s, D(N^s))$. (c) A query-graph that is directly mapped from an AQL query and will be expanded into a pattern-graph G^p. The problem is to find a sub-optimal approximate match between the pattern-graph G^p and two source-regions from $(G^s, D(N^s))$.

the pattern-region G_2^{pr} at phase 2.

Cost function

The following cost function is applied for both phases of the example matching process. At each phase a search-tree is generated where each tree-node is associated with a cost that is used for searching the tree with the employed search strategy. The applicable costs at each node of a search-tree was discussed in Section 6.5. In this example, the cost of matching a placeholder-node $n_{i,j}$ and a candidate-node n_k at each tree-node, i.e., $cost_{total}$, consists of different parts as shown below, where each underlined cost is defined in the next line:

$$\begin{aligned} cost_{total} &= \underline{cost_{path}} + cost_{uest} \\ cost_{path} &= cost_{path-1} + \underline{cost_{mch}} \\ cost_{mch} &= c_{in}^{ed} + c_{out}^{ed} + c_{out}^{ei} + c^{nd} \end{aligned}$$

These costs are defined as:

- $cost_{mch}$: cost of matching the current node with the placeholder-node which is caused by edge deletion / insertion and node deletion. This cost is further broken down into costs c_{in}^{ed}, c_{out}^{ed}, c_{out}^{ei}, and c^{nd} as discussed in Section 6.4.
  - c_{in}^{ed}: cost of deleting an edge inside the pattern-region. The cost is calculated as: $\frac{M-s}{k}$, $\frac{M-0.75s}{k}$, or $\frac{M-0.5s}{k}$, if zero, one, or two edges are deleted between the candidate-node n_k and an already matched node in the pattern-region. This cost assignment was discussed in detail in Section

6.4.2. In these formulae, M is the maximum similarity between two entities in the corresponding source-region; s is the similarity value between the candidate-node and an already matched-node in the pattern-region; and k is the number of already matched nodes in the pattern-region. In our example of Figure 6.10, s is shown in the table for domain of main-seeds as sim, and M is the maximum of sim in each table.

- c^{ed}_{out} (applicable for phase 2): cost of edge deletion from the "*imported*" edge-bundles of the pattern-region as $c^{ed}_{out} = \frac{1}{r} \times \ 0.25 \times c^{ed}_{in}$ where, r is the number of remaining edge-bundles, including the current edge-bundle. This cost assignment was discussed in Section 6.4.2. The cost of edge deletion from the *exported* edge-bundles is not applicable, since no exported query-edge is defined for query-node qn_2.
- c^{ei}_{out} (applicable for phase 2): cost of edge insertion on the *imported* edge-bundles for G^{pr}_2 is $maxCost$ since the edge-bundles are maximally generated and edge insertion exceeds the maximum number of matched edges. The cost of inserting *exported* connector-edges for G^{pr}_2 is zero, since they are not constrained by the query-graph (see Section 6.4.1).
- c^{nd}: cost of node deletion is not applicable for this example, hence it is not considered for computation of $cost_{mch}$.

• $cost_{path}$ ($cost_{path-1}$): total cost of matching the nodes at different tree-nodes along a tree-path from the root of tree to the current tree-node (parent tree-node).

- $cost_{uest}$: an under-estimation of the remaining cost from the current tree-node to a leaf tree-node. In reality, at each level of the search-tree that a new node is matched at least k inside-pattern edges are deleted (k is number of already matched nodes). We assume that the similarity value s of the rest of nodes to be matched from the current tree-node to a leaf tree-node is the highest similarity value. Also, we assume that the matched-nodes with underestimate-cost satisfy the constraints for the connector-edges, hence both costs c^{ed}_{out} and c^{ei}_{out} are zero. Therefore, the under-estimation cost for matching each remaining node, denoted as $cost^1_{uest}$, is defined using only c^{ed}_{in}, as follows:

$$cost^1_{uest} = \frac{M-0.75s}{k} \times k \qquad (\mathrm{M} = 4 \text{ and } \mathrm{s} = 4)$$
$$cost^1_{uest} = 4 - 0.75 \times 4 = 1$$

The under-estimation cost $cost_{uest}$ is defined as the total of under-estimation costs for the remaining placeholder-nodes to be matched, as:

$$cost_{uest} = \text{number of remaining placeholder-nodes} \times cost^1_{uest}$$

6.6.1 Phase 1 of matching example

Figure 6.11 illustrates phase 1 of the matching example of Figure 6.10. The pattern-graph G^p_1 consists of only the pattern-region G^{pr}_1 which is the maximal expansion (four nodes) of the query node qn_1 from the query-graph G^q. The input-graph is the source-region G^{sr}_6. The search-tree has four levels, each corresponding to matching a placeholder-node $n_{1,j}$. Each tree-node is identified by a number, cost of matching

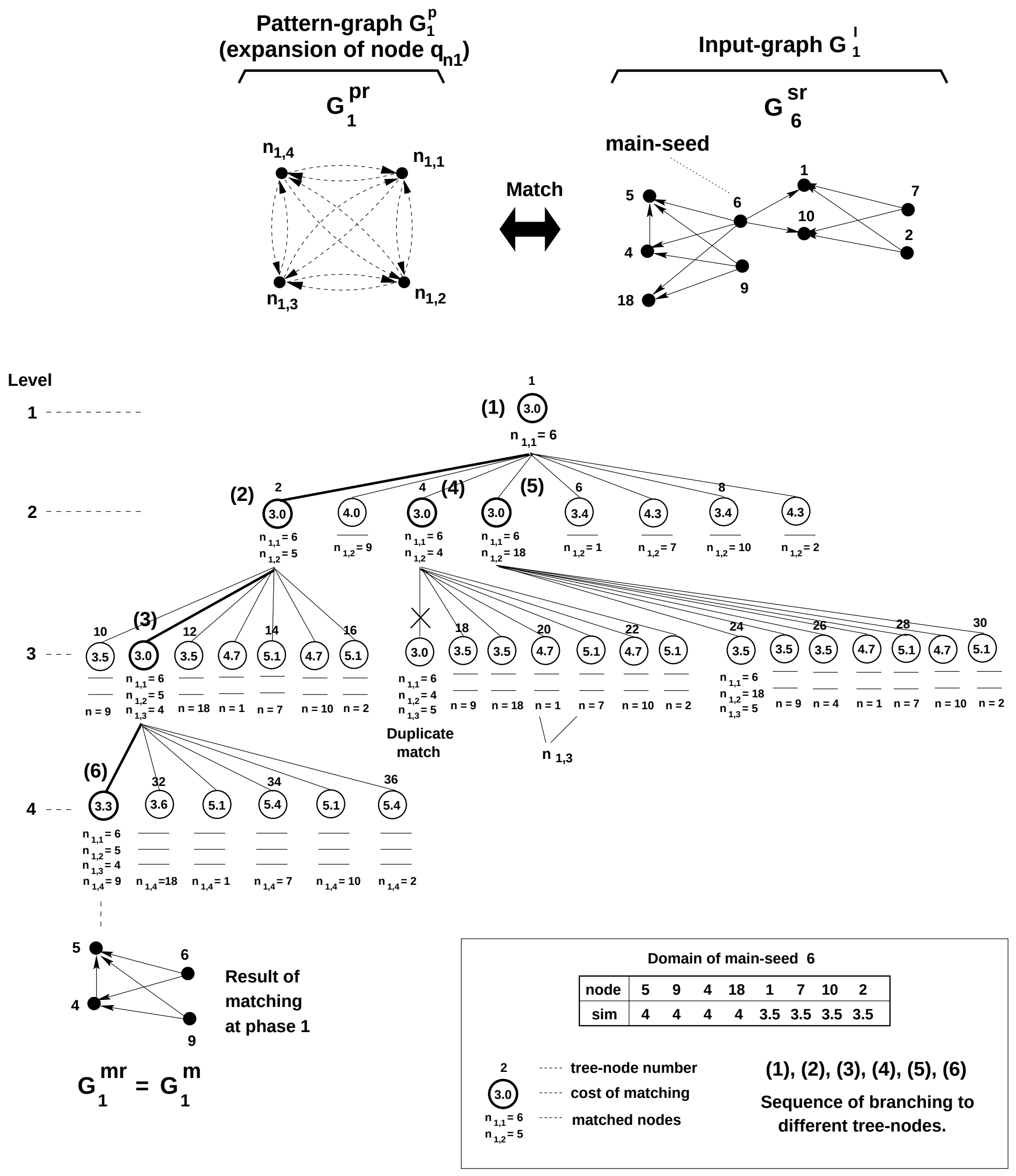


Figure 6.11: Phase 1 of the 2-phase matching example. At each tree-node (state) one node from the selected source-region G_6^{sr} is matched against one placeholder-node from pattern-graph G_1^{pr}.

a placeholder-node with a candidate-node from the source-region, and the pairs of matched nodes. The node-matching is shown as $n_{x,y} = k$, where $n_{x,y}$ and n_k are the placeholder-node and the candidate-node, respectively.

Figure 6.12 shows the cost evaluation of a number of tree-nodes in Figure 6.11. The $cost_{total}$ and its applicable elements for phase 1 are as follow:

$$\text{Phase 1:} \qquad cost_{total} = cost_{path-1} + c^{ed}_{in} + cost_{uest}$$

To clarify the notation used in the cost evaluation, we repeat below the total-cost at tree-node number 36 which is defined further in this Section.

$$c_{36_{(n_{1,4}=2,\ s=3.5,0,0,\ k=3)}} = c_{11_{path}} + c_{36_{in}} + c_{36_{uest}}$$

$c_{36(..)}$ is the abbreviation of $cost_{total}$ at tree-node number 36. The parenthesis specify the parameters for cost evaluation, including: placeholder-node $n_{1,4}$, candidate-node n_2, similarity values between the candidate-node and already matched nodes s, and number of already matched-nodes k. $c_{11_{path}}$ is $cost_{path-1}$, $c_{36_{in}}$ is c^{ed}_{in}, and finally $c_{36_{uest}}$ is under-estimation cost.

Based on the cost evaluation at different tree-nodes, the *BQ-A** algorithm expands the tree-node with lowest total cost. In Figure 6.11, the search-tree expansion is shown as the sequence of tree-nodes: 1, 2, 11, 4, 5, and 31. A thick line indicates the tree-path that produces a sub-optimal match at phase 1, i.e., the matched-region G^{mr}_1. In Figure 6.11, G^{mr}_1 consists of nodes n_6, n_5, n_4, n_9 which have the highest number of edges and average similarity value compared with any other four

$c_{1_{(n_{1,1}=6)}} = 0.0 + 0.0 + 3 \times 1.0 = 3.0$
$c_{2_{(n_{1,2}=5, s=4, k=1)}} = c_{1_{path}} + c_{2_{in}} + c_{2_{uest}}$
$c_2 = 0.0 + 1.0 + 2 \times 1.0 = 3.0$
$c_{4_{(n_{1,2}=4, s=4, k=1)}} = c_{1_{path}} + c_{4_{in}} + c_{4_{uest}}$
$c_4 = 0.0 + 1.0 + 2 \times 1.0 = 3.0$
$c_{8_{(n_{1,2}=10, s=3.5, k=1)}} = c_{1_{path}} + c_{8_{in}} + c_{8_{uest}}$
$c_8 = 0.0 + 1.4 + 2 \times 1.0 = 3.4$
$c_{11_{(n_{1,3}=4,\ s=4,4,\ k=2)}} = c_{2_{path}} + c_{11_{in}} + c_{11_{uest}}$
$c_{11} = 1.0 + 1.0 + 1.0 = 3.0$
$c_{21_{(n_{1,3}=7,\ s=3.5,0,\ k=2)}} = c_{4_{path}} + c_{21_{in}} + c_{21_{uest}}$
$c_{21} = 1.0 + 3.1 + 1.0 = 5.1$
$c_{31_{(n_{1,4}=9,\ s=4,4,4,\ k=3)}} = c_{11_{path}} + c_{31_{in}} + c_{31_{uest}}$
$c_{31} = 2.0 + 1.3 + 0.0 = 3.3$
$c_{36_{(n_{1,4}=2,\ s=3.5,0,0,\ k=3)}} = c_{11_{path}} + c_{36_{in}} + c_{36_{uest}}$
$c_{36} = 2.0 + 3.4 + 0.0 = 5.4$

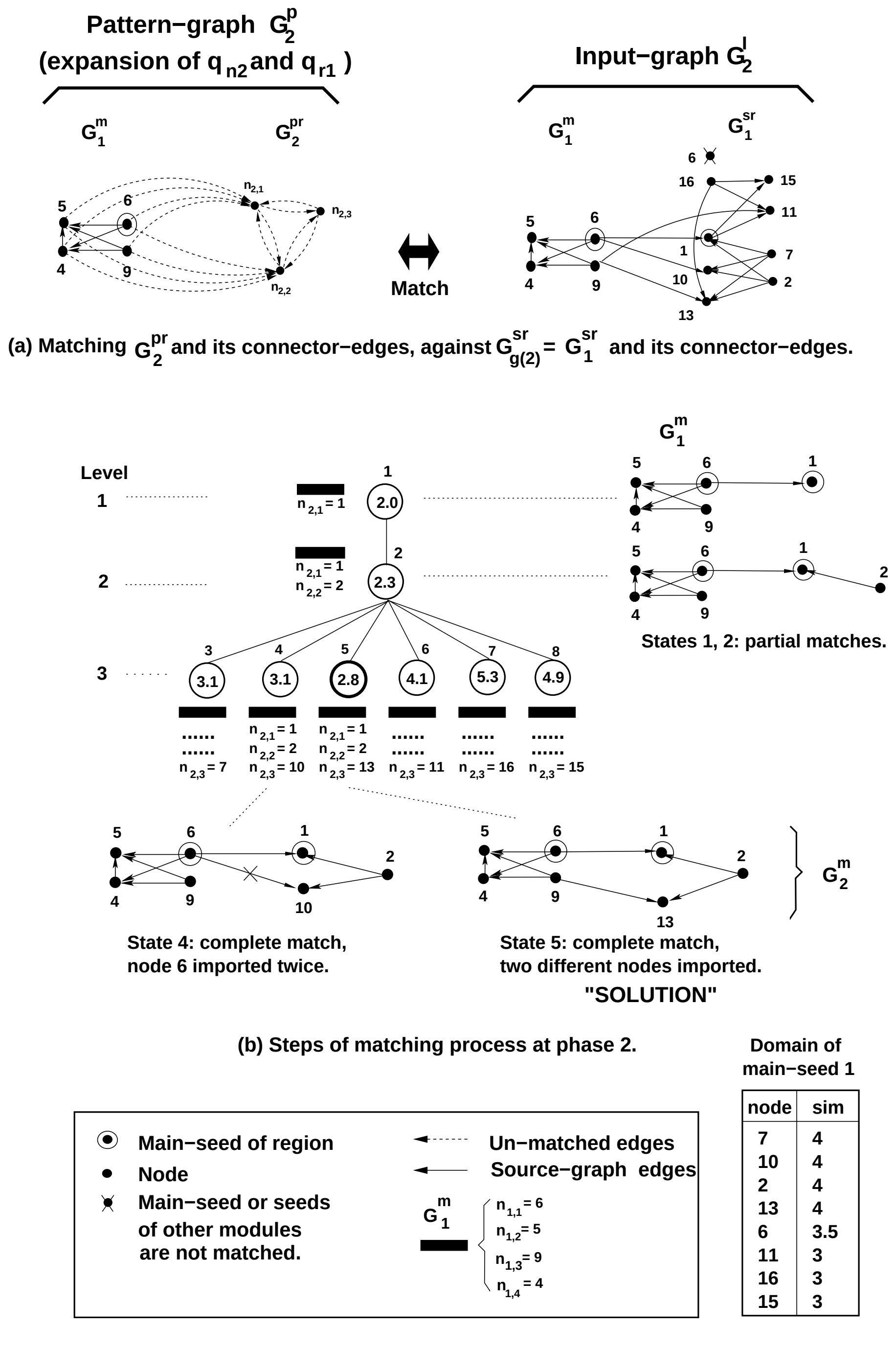
Figure 6.12: The cost evaluation for phase 1 of the matching example.

nodes in the source-region G_6^{sr}. At phase 1, the resulting matched-region G_1^{mr} is the matched-graph G_1^m.

6.6.2 Phase 2 of matching example

Figure 6.13 illustrates phase 2 of the matching example of Figure 6.10. The pattern-graph G_2^p consists of the summation of: i) matched-graph at phase 1 G_1^m; ii) pattern-region G_2^{pr} which is the expansion of the query node qn_2; and iii) connector-edges $\mathcal{R}_2^{m \leftrightarrow pr_2}$. The input-graph G_2^I consists of the summation of: i) matched-graph at phase 1 G_1^m; ii) selected source-region $G_{g(2)}^{sr} = G_1^{sr}$; and iii) connector-edges $\mathcal{R}_2^{m \leftrightarrow sr_2}$.

At phase 2, the matching algorithm incrementally matches "source-region G_1^{sr} and its connector-edges $\mathcal{R}_2^{m \leftrightarrow sr_2}$" against "pattern-region G_2^{pr} and its connector-


Pattern−graph G_2^p
(expansion of q_{n2} and q_{r1})
Input−graph G_2^I
G_1^m
G_2^{pr}
G_1^{sr}
Match
(a) Matching G_2^{pr} and its connector−edges, against $G_{g(2)}^{sr} = G_1^{sr}$ and its connector−edges.
Level
$n_{2,1} = 1$
$n_{2,2} = 2$
$n_{2,3} = 7$
$n_{2,3} = 10$
$n_{2,3} = 13$
$n_{2,3} = 11$
$n_{2,3} = 16$
$n_{2,3} = 15$
2.0
2.3
3.1
3.1
2.8
4.1
5.3
4.9
States 1, 2: partial matches.
State 4: complete match, node 6 imported twice.
State 5: complete match, two different nodes imported.
"SOLUTION"
G_2^m
(b) Steps of matching process at phase 2.
Main−seed of region
Node
Main−seed or seeds of other modules are not matched.
Un−matched edges
Source−graph edges
$n_{1,1} = 6$
$n_{1,2} = 5$
$n_{1,3} = 9$
$n_{1,4} = 4$
Domain of main−seed 1

node	sim
7	4
10	4
2	4
13	4
6	3.5
11	3
16	3
15	3

Figure 6.13: Phase 2 of the 2-phase graph-matching example. In this phase the cost of edge-deletion for both the edges that are inside the pattern-region and the connector-edges are evaluated to assign a total cost for each tree-node.

edges $\mathcal{R}_2^{m\leftrightarrow pr_2}$".

The search-tree has three levels, each corresponding to matching a placeholder-node $n_{2,1}, n_{2,2}$, and $n_{2,3}$. In Figure 6.13, the process of matching at different tree-nodes (states) of the search-tree is shown. Each tree-node contains a fixed part related to the matched-graph at phase 1 (G_1^m shown as a black rectangle) and a changing part related to matching the nodes between source-region and pattern-region.

Also, in defining the AQL query in this example, the user assigns node n_2 from source-region G_1^{sr} as a fixed node (a seed) to appear in the result of the matching at phase 2 without searching. This causes at level 2 of the search-tree only node n_2 be matched against the placeholder-node $n_{2,2}$, hence, the search-tree is pruned to exclude the matching of the remaining nodes of G_1^{sr}.

The $cost_{total}$ and its applicable elements for phase 2 (below) are obtained from the cost evaluation discussed earlier.

$$\text{Phase 2:} \qquad cost_{total} = cost_{path-1} + c_{in}^{ed} + c_{out}^{ed} + c_{out}^{ei} + cost_{uest}$$

The cost evaluation at some of the tree-nodes are shown in Figure 6.14. In this evaluation, we use the same abbreviations and $cost_{uest}$ as in phase 1. However, the sum of two costs $c_{out}^{ed} + c_{out}^{ei}$ are shown by a single cost, e.g., $c_{2_{out}}$ in Figure 6.14; and "r" (which is applicable for phases 2 and more) is defined as the number of remaining edge-bundles including the current edge-bundle, has been added to the parameter list for the cost evaluation.

At the end of the matching process, the *BQ-A** algorithm identifies the leaf node

$c_{1_{(n_{2,1}=1,\ r=2)}} = 0.0 + 0.0 + 0.0 + 2 \times 1.0 = 2.0$
$c_{2_{(n_{2,2}=2,\ s=4,\ k=1,\ r=1)}} = c_{1_{path}} + c_{2_{in}} + c_{2_{out}} + c_{2_{uest}}$
$c_2 = 0.0 + 1.0 + 0.25 + 1.0 = 2.3$
$c_{4_{(n_{2,3}=10,\ s=4,4,\ k=2,\ r=1)}} = c_{2_{path}} + c_{4_{in}} + c_{4_{out}} + c_{4_{uest}}$
$c_4 = 1.25 + 1.5 + 0.375 + 0.0 = 3.1$
$c_{5_{(n_{2,3}=13,\ s=4,4,\ k=2,\ r=1)}} = c_{2_{path}} + c_{5_{in}} + c_{5_{out}} + c_{5_{uest}}$
$c_5 = 1.25 + 1.5 + 0.0 + 0.0 = 2.8$
$c_{7_{(n_{2,3}=16,\ s=3,0,\ k=2,\ r=1)}} = c_{2_{path}} + c_{7_{in}} + c_{7_{out}} + c_{7_{uest}}$
$c_7 = 1.25 + 3.25 + 0.81 + 0.0 = 5.3$

Figure 6.14: The cost evaluation for phase 2 of the matching example.

5 with minimum graph edit cost 2.8 which generates the matched-region G_2^{mr} with three nodes and two connector-edges that import two nodes from matched-region G_1^{mr}. The resulting matched-graph G_2^m conforms with the constraints defined in the query-graph G^q in Figure 6.10. Note that the matched-graph at tree-node 4 is not the solution since two connector-edges import the same node n_6, therefore, only one connector-edge is effective.

6.7 Summary

In this Chapter, the software architecture recovery has been presented as a graph matching problem between an input-graph and a pattern-graph. The first graph originates from the software system and the second graph originates from an abstract pattern that is defined by an AQL query. The software system graph is represented as a collection of source-regions, where each region node is annotated with the similarity value to the source-region's main-seed. Similarly, the abstract pattern in the AQL query is represented as a pattern-graph with placeholders as module nodes and edge-bundles as import/export links between modules. The matching process incrementally matches the placeholder-nodes with source-region nodes and compares different alternatives of the partially-matched pattern-graphs using the bounded-queue A^* (BQ-A^*) algorithm to recover a system partition in a way that best matches with the AQL model. The matching process is based on a cost evaluation function that evaluates the graph edit operations that sub-optimally align the source-graph with the pattern-graph. The evaluated costs are set so that they ensure the link constraints in the abstract pattern of the AQL query are not violated while cohesive modules or subsystems are recovered. The use of two heuristics, one for search-space reduction and the other for path-queue size limitation of the A^* algorithm make the exponential complexity of the A^* more tractable with the penalty of obtaining a sub-optimal result instead of an optimal one.

Chapter 7

Overview of algorithms and complexity

The design of tractable algorithms for the proposed graph pattern-matching approach is the most challenging part of this work. When the size of the graphs are large, then the standard techniques to graph matching are not tractable since the computational complexity becomes very high. The major source of the computational complexity for an architectural recovery process is the employed search algorithm. Hence, in the case of using an optimal search algorithm such as A^* a trade-off between the optimality of the result and the search effort is inevitable.

In order to tackle the inherent complexity of the A^* algorithm, we utilize two heuristics: i) decomposing the whole search space into source-regions whose graph-nodes are ranked using data mining association; and ii) reducing the storage space and sorting time of the tree-paths queue using a bounded queue mechanism. The resulting algorithm, i.e., Bounded-Queue A^* (BQ-A^*) was discussed in Section 6.5.

This Chapter is organized as follows. First, the implementation view of the graphs in Chapter 5 are defined. Second, an overview of the graph pattern-matching algorithms are discussed. Finally, the computational complexity of the algorithms and the employed trade-offs are briefly discussed.

7.1 Implementation view of graphs

In order to improve the processing performance, the graphs presented in Chapter 5 are implemented as lists. In the list representation of the graphs the notation $\mathcal{G}$ (the caligraphic notation for graph G) is used. Figures 7.1(a) and (b) illustrate the different lists that are used to implement a multi-phase search-tree in Figure 7.1(d). These lists are as follows:

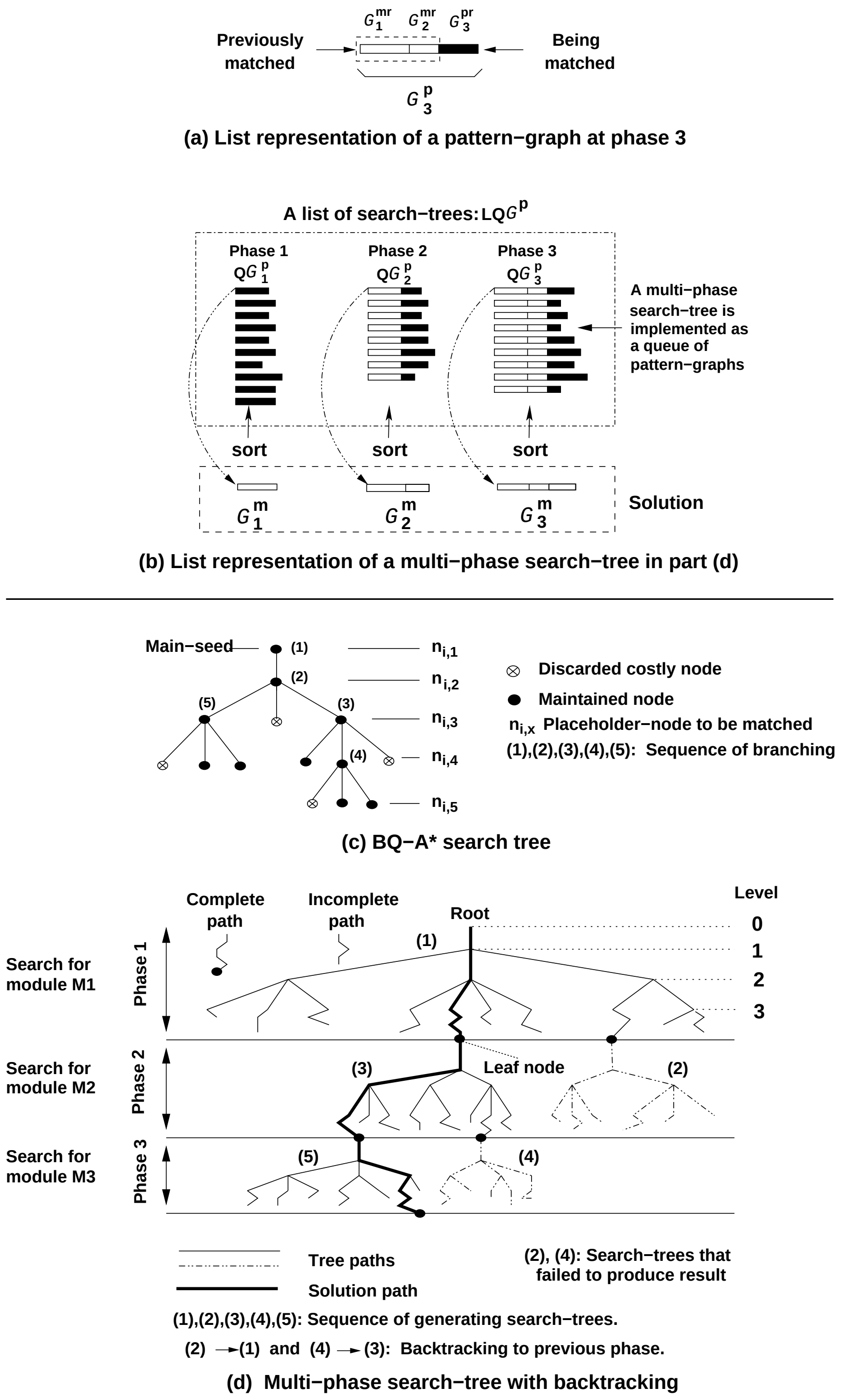
- $\mathcal{G}_i^{pr}$: a pattern-region at phase i is implemented by a list of matched node pairs, such as:
  $\mathcal{G}_i^{pr} = ([(n_{i,1}, n_x),\ (n_{i,2}, n_y),\ ...,\ (n_{i,t}, n_z)],\ \ R^s)$
  where, in $(n_{i,j}, n_k)$ $n_{i,j}$ denotes a placeholder-node from pattern-region G_i^{pr}, and n_k denotes a node from source-region $G_{g(i)}^{sr}$ that matches with $n_{i,j}$. The matched edges among the matched nodes n_x, n_y, n_z are a subset of the source-graph edges R^s, and $t < p$ where t is the size of the list and $p = |N_i^{pr}|$ is the number of the placeholder-nodes in G_i^{pr}. The pattern-region $\mathcal{G}_i^{pr}$ represents an incomplete *single-phase* tree-path.

- $\mathcal{G}_u^{mr}$: a matched-region is the same as a pattern-region $\mathcal{G}_u^{pr}$ at phase u, where all the placeholder-nodes have been matched, i.e, $t = p = |N_i^{pr}|$ in the above list. The matched-region $\mathcal{G}_u^{mr}$ represents a complete single-phase tree-path.

- $\mathcal{G}_j^{sr}$ ($\mathcal{G}_{g(i)}^{sr}$): a selected source-region at phase i is implemented by the domain D^{n_j} and the source-graph edges R^s, such as:
  $\mathcal{G}_j^{sr} = \mathcal{G}_{g(i)}^{sr} = ([(n_a, s_a),\ (n_b, s_b),\ ...,\ (n_d, s_d)],\ R^s)$
  where, $s_a = entAssoc(n_j, n_a)$. The source-region $\mathcal{G}_{g(i)}^{sr}$ represents a search domain for the nodes to be matched with the placeholders nodes $\mathcal{G}_i^{pr}$.

- $\mathcal{G}_i^p$: a pattern-graph at phase i is implemented as a list of $i-1$ matched-regions and a pattern-region at the end of the list, such as:
  $\mathcal{G}_i^p = [\ (\mathcal{G}_u^{mr},\ \mathcal{R}_u^{mr^* \leftrightarrow mr_u})\ |\ u \in [1\ ..\ i-1]\ ]\ \ concat\ \ [\ (\mathcal{G}_i^{pr},\ \mathcal{R}_i^{mr^* \leftrightarrow pr_i})\ ]$
  where $\mathcal{R}_i^{mr^* \leftrightarrow mr}$ represents all the matched connector-edges between every matched-region (shown by mr^*) and the matched-region $\mathcal{G}_u^{mr}$; and $\mathcal{R}_i^{mr^* \leftrightarrow pr_i}$ represents all the connector-edges (i.e., both edge-bundles and matched connector-edges) between every matched-region (shown by mr^*) and the pattern-region $\mathcal{G}_i^{pr}$. A pattern-graph $\mathcal{G}_i^p$ represents an incomplete *multi-phase* tree-path consisting of zero or more complete tree-paths for the matched-regions and one incomplete tree-path for the pattern-region. In Figure 7.1(d) every tree-path from the root to the end of a tree-path in different phases (except the thick path) is an incomplete multi-phase tree-path, whose list implementation is shown in Figures 7.1(a).

- $\mathcal{G}_i^m$: a matched-graph at phase i is a pattern-graph $\mathcal{G}_i^p$ where all the placeholder-nodes in the pattern-region $\mathcal{G}_i^{pr}$ have been matched. In other words, $\mathcal{G}_i^m$ represents a complete multi-phase tree-path. In Figure 7.1(d) the thick line



Previously matched
Being matched
G_1^{mr} G_2^{mr} G_3^{pr}
G_3^p
(a) List representation of a pattern−graph at phase 3
A list of search−trees: LQG^p
Phase 1 QG_1^p
Phase 2 QG_2^p
Phase 3 QG_3^p
A multi−phase search−tree is implemented as a queue of pattern−graphs
sort
Solution
G_1^m G_2^m G_3^m
(b) List representation of a multi−phase search−tree in part (d)
Main−seed
$n_{i,1}$ $n_{i,2}$ $n_{i,3}$ $n_{i,4}$ $n_{i,5}$
⊗ Discarded costly node
● Maintained node
$n_{i,x}$ Placeholder−node to be matched
(1),(2),(3),(4),(5): Sequence of branching
(c) BQ−A* search tree
Complete path
Incomplete path
Root
Level 0 1 2 3
Search for module M1
Search for module M2
Search for module M3
Phase 1
Phase 2
Phase 3
Leaf node
Tree paths
Solution path
(2), (4): Search−trees that failed to produce result
(1),(2),(3),(4),(5): Sequence of generating search−trees.
(2) → (1) and (4) → (3): Backtracking to previous phase.
(d) Multi−phase search−tree with backtracking

Figure 7.1: (a) A pattern-graph $\mathcal{G}_i^p$ as a list of its graph elements. (b) A *BQ-A** search-tree is implemented as a queue of pattern-graphs $Q\mathcal{G}_i^p$. Maintaining a list of multi-phase search-trees $LQ\mathcal{G}^p$ allows to implement backtracking at module-level.

from the root to a leaf node at phase 3 is a matched-graph or a recovered architecture.

- $Q\mathcal{G}^p_i$: a multi-phase search-tree at phase i in Figure 7.1(d) is implemented as a queue of incomplete multi-phase tree-paths (i.e., partially-matched pattern-graphs) in Figure 7.1(b), such as:
  $Q\mathcal{G}^p_i = [\ \mathcal{G}^p_{i_x} \mid x \in [1, 2, \ ...]\ ]$

- $LQ\mathcal{G}^p$: a list of multi-phase search-trees at different phases is implemented as a list of queues of incomplete multi-phase tree-paths, as illustrated in Figure 7.1(b). $LQ\mathcal{G}^p$ allows to implement a backtracking mechanism at the module level, where:
  $LQ\mathcal{G}^p = [\ Q\mathcal{G}^p_k \mid k \in [1 \ .. \ i]\ ]$

To simplify the description of the algorithms, in the remaining of this Section every "search-tree" is a "*multi-phase search-tree*" and every "tree-path" is a "*multi-phase tree-path*".

7.2 Overview of algorithms

In this Section, an overview of the algorithms that collectively implement the proposed graph pattern-matching algorithms is provided. The pseudocode of these algorithms is presented in Appendix C. Also, Figure 7.2 illustrates a summary of the algorithms and a flowchart that sketches the sequence of algorithm invocations. The first two algorithms provide the top-level control mechanism to handle different types of entities as well as backtracking between phases. The other six algorithms implement a *BQ-A** search algorithm for a sub-optimal graph-matching based on graph-edit costs.

1. **Algorithm** *main-analysis* (*AQLquery, S*)

   The main-analysis algorithm, as the top-level control mechanism, essentially handles the semantic checking of the AQL query and controls the analysis as discussed in Section 4.2.4. The operations of this algorithm are as follows: i) invoking the AQL parser to parse the AQL query in the file *AQLquery*; ii) controlling the analysis process for each single entity-type in the query (i.e., *Function-abs, Type-abs*, and *Variable-abs*) and accumulating the result of recovery for each entity-type; iii) resolving the shared entities in the result of recovery; and iv) presenting the generated solution for the system architecture through HTML pages and graphs to be visualized.

2. **Algorithm** *control-iterative-recovery* ($\mathcal{G}^q, entType, S$)

   The second-level control mechanism performs an iterative matching-process at different phases $i \in [1..|N^q|]$ for the recovery of modules that contain only one entity-type (i.e., *Function-abs / Type-abs / Variable-abs*). This control mechanism operates on the collection of the stored multi-phase search-trees in the list $LQ\mathcal{G}^p$ that are generated by the *BQ-A** search algorithm at different phases. The multi-phase search-tree at phase i, i.e, $Q\mathcal{G}^p$, is in the form of a queue of incomplete search-paths. The control mechanism is responsible for supervising the matching process at module level, by: i) retrieving a multi-phase search-tree either from the list $LQ\mathcal{G}^p$ (if it is already generated) or by creating it from the AQL query information that exist in the query-graph $\mathcal{G}^q$;

```
1  main−analysis (AQLquery, S)

% S represents the 'source−graph' and 'domains of nodes'
− "Gq" := Parse "AQL query" and check its semantics
− LOOP  for every "entType": F / T / V    DO
      − "Gm" := control−iterative−recovery using .....  2
      − "Gm" := resolve−shared−entities ("Gm")
      − "solution" := accumulate resuling "Gm" for F / T / V
− "HTML pages & Graphs" := generate HTML pages
   and Rigi graphs from the "solution"
− output  "HTML pages & Graphs"
− STOP
```

```
2  control−iterative−recovery (Gq, entType, S)

% Control the iterative generation of the architectural solution.
% All computations are based on current "entType" F or T or V.
− LOOP for every module in "Gq" till 'all fail' or 'all success'  DO
    − "QGp" := retrieve search−tree from global list "LQGp"
    − IF no "QGp" for this phase, THEN create one using .....  3
    − "Gm" := search "QGp" for "Gm" using BQ−A*  in ........  4
    − IF "Gm" found in the last phase THEN 'all  success'
    − IF "Gm" found in a middle phase THEN go to next phase
    − IF no "Gm" found THEN Backtrack to previous phase
    − IF no "Gm" found in the first phase  THEN 'all fail'
    − restore partially expanded "QGp" to global list "LQGp"
− RETURN "Gm"
```

```
4  BQ−A*  (QGp, Gsr', i, S)

− WHILE  search−tree "QGp" is not empy  OR
                                 "Gm" not found   DO
  − "Gp" := remove multi−phase tree−path from head of  "QGp"

  − "Gpr" := get single−phase tree−path from "Gp"
  − IF "Gpr" is fully−matched   THEN
      − "Gm" := "Gp"  with "Gpr" in "Gp",  and  exit the loop
  − ELSE
      − "ph" := get the current placeholder−node  from "Gpr"
      − FOR  every region−node "nd" in "Gsr'"  DO
          − IF  new tree−path with "nd" is not duplicate THEN
              % perform node / edge  matching for "ph = nd"
              % evaluate graph−edit cost for match "ph = nd"
              − "Gp" := restore "Gpr" in  "Gp"
              − ("Gp", "cost") := evaluate−node−matching−cost
              − IF "cost" < "maxCost" THEN                5
                  − "QGp" := insert new "Gp" into queue  and sort it
− RETURN   ("Gm",  "QGp")
```

```
3  create−and−initialize−tree (Gq, Gm, i, S)

− "Gp" := generate and initialize a tree−path using "Gq" and "Gm"
− "QGp" := create a single−path search−tree for BQ−A* using "Gp"
− "Gsr'" := get elligible region−nodes of "Gsr" as search−space
− RETURN ("QGp", "Gsr'")
```

```
6  inside−edge−deletion−cost (Gp, nd, i, S)

− FOR  every matched "node" in the matching−pairs of "Gpr"  Do
   % test edges between "node" and "nd"
   − "cost" := cost for deleting 0 / 1 / 2 edges using similarity
   − "Cin" : = sum of "cost"
− RETURN "Cin"
```

```
5  evaluate−node−matching−cost (Gp, nd, i, S)

− "Gp" := insert matching−pair  ("ph", "nd") into tree−path "Gp"
− "Cin" := get inside−edge−deletion−cost  using ...  6
− "Cimp" := get import−edge−matching−cost using .......  7
− "Cexp" := get export−edge−matching−cost  using ...  8
− "cost" := get total of "Cin", "Cimp", "Cexp"
− RETURN   ("Gp", "cost")
```

```
7  import−edge−matching−cost (Gp, nd, i, S)
8  export−edge−matching−cost (Gp, nd, i, S)

− For every link−module "Gmr" in "Gp" that is linked to "Gpr"   DO
    − check edge−bundles between "Gpr" and  "Gmr"
    − get existing edges between "Gpr" and "Gmr" in source−graph
    − IF existing edges violate Min or Max threshold THEN
        − "Cimp" / "Cexp" := maxCost    % exit loop
    − IF existing edges are within range THEN
        − generate matched edges and add to connector−edges
        − "Cimp" / "Cexp" := total cost of edge deletion / insertion
    − perform consistency checking to delete the duplicate
      un−linked nodes in other modules.
− RETURN  "Cimp" / "Cexp"
```

$Gq = G^{q}$

$Gp = G^{p}_{i}$

$Gm = G^{m}_{i}$

$Gsr = G^{sr}_{g(i)}$

$Gpr = G^{pr}_{i}$

$Gmr = G^{mr}_{u}$

$QGp = QG^{p}_{i}$

$LQGp = LQG^{p}$

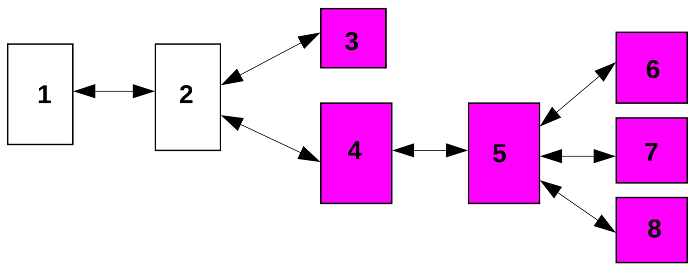


Figure 7.2: A summary of algorithms for architectural recovery based on graph pattern matching. The flow-chart indicates the sequence of invocation, where the first two algorithms provide control mechanism and the remaining algorithms perform the pattern matching operation.

ii) invoking the *BQ-A** search algorithm to search and expand the search-tree $Q\mathcal{G}_i^p$ and produce a solution as the matched-graph $\mathcal{G}_i^m$; and iii) backtracking to the previous phase $i-1$ using the stored multi-phase search-tree $Q\mathcal{G}_{i-1}^p$ in the list $LQ\mathcal{G}^p$ in order to find another solution for the module $\mathcal{G}_{i-1}^m$ and then return to the current phase i. Figure 7.1(d) illustrates the sequence of generating the multi-phase search-trees and backtracking to the previous phases.

3. **Algorithm** *create-and-initialize-tree* $(\mathcal{G}^q, \mathcal{G}_i^m, i, S)$

   This algorithm generates a multi-phase search-tree $Q\mathcal{G}_i^p$ with a single-path for phase i to be explored by the *BQ-A** search algorithm. The generated search-tree contains a single incomplete-path $\mathcal{G}_i^p$ (i.e., a pattern-graph) where the tree-nodes of its $\mathcal{G}_i^{pr}$ have been initialized with matching pairs of nodes for main-seeds and seeds (i.e., one or more placeholder-nodes from pattern-region $\mathcal{G}_i^{pr}$ in $\mathcal{G}_i^p$ have been matched). For example, in Figure 6.13 at phase 2, the tree-nodes 1 and 2 have been generated in this algorithm. The algorithm also prepares the search-space for the *BQ-A** search algorithm by excluding specific nodes, such as: main-seeds, seeds, and already matched-and-linked nodes, from the selected source-region $\mathcal{G}_{g(i)}^{sr}$. The resulting search-space is denoted as $\mathcal{G}_{g(i)}^{sr'}$.

4. **Algorithm** *BQ-A** $(Q\mathcal{G}_i^p,\ \mathcal{G}_{g(i)}^{sr'},\ i,\ S)$

   The bounded-queue A^* search algorithm (*BQ-A**) iterates in a while loop until either all the incomplete tree-paths $\mathcal{G}_i^p$ (partially-matched pattern-graphs) in the search-tree $Q\mathcal{G}_i^p$ are exhausted (i.e., search failed), or a complete tree-path $\mathcal{G}_i^m$ is generated (i.e., a sub-optimal solution has been found). The generated matched-graph $\mathcal{G}_i^m$ is returned as the solution for phase i. In the loop, the algorithm first removes the lowest-cost partially-matched pattern-graph $\mathcal{G}_i^p$ from the head of queue $Q\mathcal{G}_i^p$ and obtains the current pattern-region $\mathcal{G}_i^{pr}$ to work on it. If all the placeholder-nodes in $\mathcal{G}_i^{pr}$ have already been matched then a solution has been found and the search is over for this phase. Otherwise, a number of new partially-matched pattern-graphs are generated by matching a new placeholder-node "ph" from $\mathcal{G}_i^{pr}$ with a remaining source-region node "nd" from $\mathcal{G}_j^{sr'}$. The new partially-matched $\mathcal{G}_i^{pr}$ must not be repeated, otherwise it is discarded. This duplication detection is achieved by a *History* checking and updating mechanism. Then the graph edit cost for each matching-pair (ph, nd) is evaluated and if the *cost* (i.e., $cost_{total}$) is less than $maxCost$ then the corresponding pattern-graph $\mathcal{G}_i^p$ is inserted into the queue $Q\mathcal{G}_i^p$ and the queue is sorted. Otherwise, the new $\mathcal{G}_i^p$ is discarded as a costly graph. *maxCost* occurs when the minimum (maximum) number of matched connector-edges are not reached (are exceeded).

5. **Algorithm** *evaluate-node-matching-cost* ($\mathcal{G}_i^p$, n_d, i, S)

   This algorithm performs the actual matching of two nodes "ph" and "nd" in pattern-region $\mathcal{G}_i^{pr}$ as a part of $\mathcal{G}_i^p$. In doing so, three algorithms are invoked to evaluate inside-edge deletion cost (c_{in}) and connector-edge deletion/insertion cost for imported and exported connector-edges (c_{imp} and c_{exp}). The resulting $\mathcal{G}_i^p$ and the total cost are returned.

6. **Algorithm** *inside-edge-deletion-cost* ($\mathcal{G}_i^p$, n_d, i, S)

   This algorithm applies the cost function for inside-edge deletion based on two criteria: i) number of edges that exist between the candidate-node "*nd*" and each individual node "*node*" already matched in $\mathcal{G}_i^{pr}$, and ii) similarity values between the corresponding nodes, which are intended to produce a cohesive module. This cost was discussed in detail in Section 6.4.1.

7. & 8. **Algorithms** *import- / export-edge-matching-cost* ($\mathcal{G}_i^p$, n_d, i, S)

   These two algorithms evaluate the cost of matching edge-bundles $\mathcal{R}_i^{m\leftrightarrow pr_i}$ (i.e., $\mathcal{R}_i^{mr^*\leftrightarrow pr_i}$) that connect matched-graph $\mathcal{G}_{i-1}^m$ to pattern-region $\mathcal{G}_i^{pr}$ against the connector-edges $\mathcal{R}_i^{m\leftrightarrow sr_i}$ that connect $\mathcal{G}_{i-1}^m$ to source-region $\mathcal{G}_{g(i)}^{sr}$. This matching is done in a loop that checks the edge-bundles between pattern-region and each of already recovered module $\mathcal{G}_u^{mr}$. For each linked-module $\mathcal{G}_u^{mr}$, the actual import/export connector-edges $\mathcal{R}_i^{mr_u\leftrightarrow sr_i}$ are obtained by examining the edges in source-graph G^s and if possible they are matched with the edge-bundles in $\mathcal{R}_i^{mr_u\leftrightarrow pr_i}$. For each node-matching, three cases are checked: i) and ii) if the number of matched connector-edges are less than minimum or more than maximum then the cost is *maxCost* which causes the new pattern-graph $\mathcal{G}_i^p$ (i.e., tree-path) to be discarded; and iii) the number of matched connector-edges are within the specified range, where c_{imp}/c_{exp} is evaluated according to the number of matched edges. For the import part, the cost c_{imp} of matching a pair of nodes (ph, n_d) is evaluated based on the success of node n_d in augmenting the number of imported connector-edges to reach to its maximum number. Different cases for deleting/inserting edges from imported edge-bundles or exported edge-bundles are presented in Figures 6.7 and 6.8, respectively.

Whenever the connector-edges are matched, a consistency checking operation is performed to delete the unlinked-nodes in other recovered modules that are the same as the nodes that are just linked (i.e., repeated nodes).

7.3 Complexity analysis overview

All the complexity analysis has been performed based on the worst case running time. However, the average running time of the *BQ-A** algorithm is very promising and is close to its best case running time (as in the A^* algorithm), provided that a good underestimation cost function is used. This allows to analyze middle size software systems in a tractable process.

In this section, first the implementation details of the connector-edges $\mathcal{R}_i^{mr^* \leftrightarrow pr_i}$ (between the pattern-region $\mathcal{G}_i^{pr}$ and the linked-modules $\mathcal{G}_u^{mr}$'s) and the matrix representation of the source-graph G^s are discussed. Second, an overview of the computational complexity of the graph matching algorithms in section 7.2 (using the pseudocode of the algorithms in Appendix C) is provided. Finally, a trade-off between the optimality and performance of the proposed approach is discussed.

The complexity analysis starts from the algorithms for matching imported/exported connector-edges and inside-edges. Then, the algorithms *BQ-A** search and iterative-recovery are considered.

7.3.1 Implementation of connector-edges

The implementation of the connector-edges is crucial in reducing the complexity of the 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 representation; and iii) accessing the information about an edge in time $O(1)$.

- $\mathcal{R}_i^{mr_u \rightarrow pr_i|_{bdl}}$ ($\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}}$): a group of imported edge-bundles $bi_1, bi_2, ..., bi_{i_u}$ (or exported edge-bundles $be_1, be_2, ..., be_{e_u}$) between the pattern-region $\mathcal{G}_i^{pr}$ and the link-module $\mathcal{G}_u^{mr}$ is simply represented as a positive integer i_u (or similarly e_u) that indicates the first i_u (or e_u) placeholder-nodes $n_{i,j}$'s as the sink (or source) nodes of the corresponding edge-bundles. Therefore, deleting a whole edge-bundle or matching a part of edges in an edge-bundle simply means decrementing the integer i_u (or e_u) by one; and redirecting an edge-bundle means no change on this value. These operations are performed in constant time $O(1)$.

- $\mathcal{R}_i^{mr^* \leftrightarrow pr_i}$ ($\mathcal{R}_i^{mr^* \leftrightarrow mr_u}$): the group of "all connector-edges" to the pattern-region $\mathcal{G}_i^{pr}$ (or matched-region $\mathcal{G}_u^{mr}$) is implemented as a tuple ($\mathcal{R}_i^{mr^* \rightarrow pr_i}$, $\mathcal{R}_i^{mr^* \leftarrow pr_i}$) to separate the collection as imported and exported connector-edges. In this form, $\mathcal{R}_i^{mr^* \rightarrow pr_i}$ (or similarly $\mathcal{R}_i^{mr^* \leftarrow pr_i}$) is implemented as a list of tuples:

$$\begin{aligned}\mathcal{R}_i^{mr^* \rightarrow pr_i} = [i_u;\ E_u \mid\ & u \in [1 \,..\, i-1] \ \wedge \ (\ (i_u, E_u) = \text{NIL} \ \vee \\ & (i_u \geq 0 \ \wedge \ E_u = \mathcal{R}_i^{mr_u \rightarrow pr_i|_{mch}} = \\ & \{(n_{src}, n_k) \mid n_{src} \in N_u^{mr} \ \wedge \ n_k \in N_{i_{mch}}^{pr}\})\ )\ ]\end{aligned}$$

where, each list entry is either a NIL tuple or a tuple consisting of the number of edge-bundles and the list of matched connector-edges. Therefore, at phase i accessing to the connector-edge information of each link-module $\mathcal{G}_u^{mr}$ is performed in constant time $O(1)$ by accessing a list entry that is indexed by the link-module's id-number, e.g., u. This implementation conforms with the domain model of the AQL presented in section 4.2.1.

- $M(G^s)$ *matrix of source-graph*: the source-graph $G^s = (N^s, R^s)$ is a central artifact in the proposed graph matching process. Fast testing the presence/absence of an edge between two graph-nodes n_k and n_l, and also accessing the similarity value between two nodes, i.e., $entAssoc(n_k, n_l)$ is crucial to the performance characteristics of the matching process. For this reason, the source-graph G^s is implemented as a matrix $M(G^s)$:

$$\begin{aligned} M(G^s) = [row_i \mid row_i = [a_{i,j} \mid \forall\, i,j \in [1\,..\,|N^s|] \;\bullet\; a_{i,j} = (e_{i,j}, s_{i,j}) \;\wedge \\ if\;\; (n_i, n_j) \in R^s \;\; then\;\; (e_{i,j}, s_{i,j}) = (1,\; entAssoc(n_i, n_j)) \\ else\;\; (e_{i,j}, s_{i,j}) = (0,\; entAssoc(n_i, n_j))]\;]. \end{aligned}$$

If the graph nodes n_i and n_j are of type *File-abs* then the similarity measure $entAssoc(n_i, n_j)$ is replaced by the $groupAssoc(n_i, n_j)$ measure. The matrix $M(G^s)$ allows to test the edge between two nodes and obtain the similarity value between two nodes in constant time $O(1)$ since elements are accessed by index position. The matrix $M(G^s)$ is generated *off-line* and is kept in a database to be used for the matching process. The space complexity of matrix $M(G^s)$ is $O(n^2)$, where $n = |N^s|$.

The complexity analysis is performed based on the following quantities:

$n = |N^s|$ number of nodes in the source-graph G^s.

$s = |N^{sr}_{g(i)}|$ number of nodes in the selected source-region $\mathcal{G}^{sr}_{g(i)}$.

$p = |N^{pr}_i|$ number of nodes in the pattern-region or any matched-region[1].

$q = |N^q|$ number of modules to be recovered in the whole pattern-graph $\mathcal{G}^p$.

$k =$ number of operations with running time $O(1)$ in an algorithm.

7.3.2 Computational complexity of algorithms

The complexity analysis is performed with respect to the overview of the algorithms in Figure 7.2 and the pseudocode of the algorithms in Appendix C.

Complexity: import-/ export-edge matching cost: the operations of this algorithm (corresponding to boxes 7 and 8 in Figure 7.2) are enclosed in a loop that checks the imported/exported connector-edges from every link-module $\mathcal{G}^{mr}_u$ to the pattern region $\mathcal{G}^{pr}_i$, where the upper-bound of the loop iteration is $O(q)$. The costly operations belong to obtaining the source nodes of the connector-edges in the link-module $\mathcal{G}^{mr}_u$ that are imported by the currently matched node, but are not already imported by the previously matched nodes at phase i. Figure 7.3 highlights the sets of "not already imported" source-nodes N^{src}_u in the link-module at matching phase i. The sets of nodes N^{src}_u are used to generate new imported matched connector-edges between the link-module and the pattern-region, that take the time $O(sp)$, as discussed in the

[1] p is approximately 10 for a subsystem of files, and approximately 20 for a module of simple entities.

footnote[2]. This complexity has been achieved because of the pre-processed data as baskets of items in $B(G^s)$ that store the source/sink nodes of the connector-edges to every node in the source-graph G^s. The set of sink-nodes N_u^{sink}, that is required for generating the matched exported connector-edges, can already been exported to (as illustrated in Figure 7.3(a)). However, computing N_u^{sink} has the same complexity as computing N_u^{src}, i.e., time $O(sp)$.

The implementation of the edge-bundles (as an integer number) simplifies the cost evaluation for deletion/insertion of the edge-bundles as decrementing a number from an integer in time $O(1)$. Therefore, after computing the set of source-nodes N_u^{src} (or sink-nodes N_u^{sink}), generating the matched connector-edges and adding them to the list of matched connector-edges $\mathcal{R}_i^{mr_u \leftrightarrow pr_i|_{mch}}$ in both linked-module and pattern-region is performed in a loop with $|N_u^{src}|$ (or $|N_u^{sink}|$) iterations, and in time $O(p)$.

After generating the imported/exported connector-edges for each module, a *consistency* operation is performed on $\mathcal{G}_i^p$ to delete the other instances of the linked nodes, n_{src} (or n_{sink}) and n_k so that the linked nodes become unique in

[2] $N_u^{src} = \{n_{src} \mid n_{src} \in N_u^{mr} \;\wedge\; \exists n_k \;\in\; N_{i_{mch}}^{pr} \;\bullet\; (n_{src}, n_k) \in R^s \;\wedge\; \forall (n_x, n_y) \in \mathcal{R}_i^{mr_u \rightarrow pr_i|_{mch}} \;\bullet\; n_{src} \neq n_x\}$
All the imported connector-edges for a node n_k, i.e., $(n_{src}, n_k) \in R^s$ have already been extracted in the form of a basket n_k of items n_{src}'s in $B(G^s)$ (data mining in section 3.2.2). Therefore, in computing N_u^{src} two loops are executed: in the first loop, every source-node n_{src} should be compared against all nodes N_u^{mr} to select those source-nodes that exist in the corresponding link-module; in the second loop, every selected source-node should be compared against all source-nodes of the matched connector-edges $\mathcal{R}_i^{mr_u \rightarrow pr_i|_{mch}}$ to exclude the already imported source-nodes. These operations take the time $O(sp) + O(p^2) = O(sp)$ since $s \geq p$.

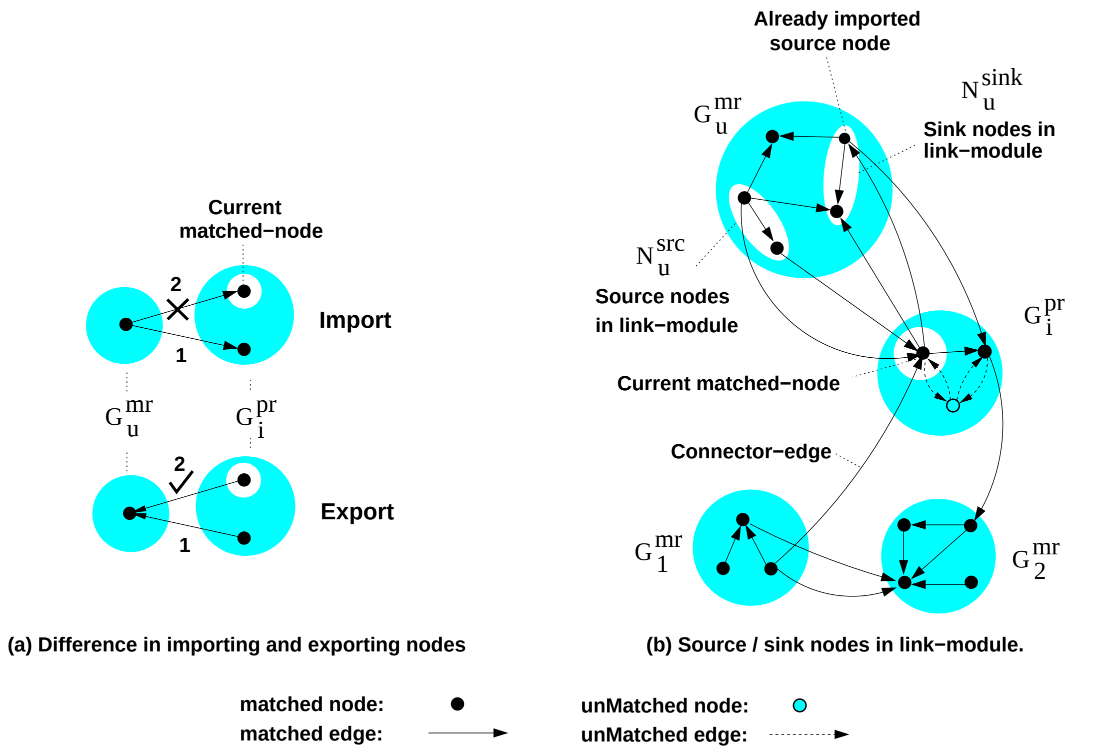


Figure 7.3: (a) Importing the same node more than once from the link-module $\mathcal{G}_u^{mr}$ is not correct (repeated import), whereas, exporting more than one node to the same node in the link-module is correct. (b) The second node (current matched-node) has been matched, and the set of (not previously imported) source-nodes N_u^{src}, and the set of sink-nodes N_u^{sink} in the linked-module are highlighted.

recovery result. This operation takes the time $O(qp)$ for each module $\mathcal{G}_u^{mr}$.[3] Therefore, the overall complexity of matching the imported (or exported) connector-edges for all q linked-modules, as a result of matching the current node n_k, can be simplified as: $O((sp + p + qp) * q) = O(qsp)$ since p can be neglected and $q \ll s$.

[3]Since the previously imported source-nodes can not be imported again, we can assume that "on average" at each node-matching at most one source-node can be imported.

Complexity: evaluate node-matching cost: the time complexity of this algorithm (corresponding to box 5 in Figure 7.2) is the total complexity of three algorithms: *inside-edge-deletion-cost*, *import-edge-matching-cost*, and *export-edge-matching-cost*. In the first algorithm, in a loop with iteration $O(p)$ all the already matched nodes *"node"* in the pattern-region $\mathcal{G}_i^{pr}$ are tested against the current node *"nd"* for possible edges and obtaining the in-between similarity values. The access to the information about each edge is performed in $O(1)$ by looking-up the source-graph matrix $M(G^s)$. Therefore, the total time complexity of the evaluate node-matching cost, using the complexity analysis of the above three algorithms is $O(p + qsp + qsp) = O(qsp)$.

Complexity: BQ-A^***:** the BQ-A^* algorithm (corresponding to box 4 in Figure 7.2) generates the search-tree by expanding the tree-nodes. The search-tree has "p" levels and each tree-node is expanded (with branching factor "s") to yield "s" new tree-nodes, each corresponding to a new incomplete tree-path that is put in the bounded-queue of tree-paths $Q\mathcal{G}_i^p$ with maximum size "b".

In the worst case running time of an A^* algorithm all tree-paths of the search-tree are expanded. In this case, all the tree-nodes of the search-tree have been visited and the total number of tree-paths in $Q\mathcal{G}_i^p$ are measured as follows: i) at each level k the number of all the expanded tree-paths is $O(s^k)$; ii) at the last level p the number of tree-paths is $O(s^p)$; iii) since the search-tree has p levels then the total number of tree-paths (or equivalently the number of tree-nodes to be visited) is $O(ps^p)$.

In the proposed BQ-A^* the number of tree-paths in the bounded-queue is maintained at size b, therefore, a large number of tree-paths are pruned and never get a chance to be expanded. This causes the number of expanded tree-paths in the bounded-queue be a sub-exponential number.

According to *BQ-A** in Figure 7.2, for each expanded tree-path three major operations are performed as follows:

- The *BQ-A** algorithm checks for the repeated *states* in the search-tree that have already been encountered and their costs of node-matching have been evaluated. A *state* represents the set of matched-nodes in a tree-path. The problem of repeated states is inherent to the informed search algorithms. For efficiency considerations, all the repeated states (tree-paths) are usually stored in a hash-table. Since the states are only inserted in the hash-table and searched (no deletion), hence the use of a proper hashing mechanism such as *double hashing* with *open addressing* [29] greatly enhances the performance. In this analysis we assume that the hash-table required time $O(h)$ for duplicate checking.
- evaluating the cost of graph node-matching takes the time $O(qsp)$, as discussed earlier.
- The *BQ-A** algorithm uses *insertion sort* to sort the tree-paths in the queue $Q\mathcal{G}_i^p$ according to the *cost* of node-matching. Since before each insertion the queue is already sorted, then the sorting requires time $O(b)$.

Therefore, the total complexity of the *BQ-A** algorithm is $O(ps^p(h+qsp+b))$.

Complexity: control-iterative recovery: The algorithm (corresponding to box

2 in Figure 7.2) consists of a *while* loop that allows to go back and forth between different phases of a multi-phase search-tree $Q\mathcal{G}_i^p$ that are stored in the list $LQ\mathcal{G}^p$. The two major operation of this algorithm are: i) invoking the algorithm *create-and-initialize-tree* which generates a multi-phase search-tree for phase i with a single tree-path for matching the main-seeds and seeds of the current module that take time $O(qps)$; and ii) invoking the $BQ\text{-}A^*$ algorithm for each phase i to generate a matched-graph G_i^m which at worst case runs in time $O(ps^p(h+qsp+b))$. Therefore, the worst case running time complexity of the proposed architecture recovery is $O(q) \times O(ps^p(h+qsp+b)+qps)$ which is simplified as $O(qps^p(h+qsp+b))$.

In the best case performance, the $BQ\text{-}A^*$ algorithm expands exactly one tree-path on each level of the search-tree. Since there are "p" levels and each tree-path can be expanded into maximally "s" successor tree-paths, the total number of tree-paths in the search-tree will be limited to $O(ps)$. Given a good underestimation cost for the remaining graph node-matching in $\mathcal{G}_i^{pr}$ improves the performance of the $BQ\text{-}A^*$ and saves the number of expanded states.

7.4 Trade-off, optimality vs. performance

The search techniques play an important role in exploring non-trivial relationships in a software system as a part of a reverse engineering task. Because of the prohibitive size of the search space in dealing with large systems, it is imperative to aim for a trade-off between quality and search complexity. In this context, some researchers use non-complete and non-optimal but fast search techniques such as *hill climbing* [72]. In this thesis, the focus is on using an optimal search technique such as A^* and limiting its exponential time and space complexity at worse case using two heuristics. The resulting version of A^* is tractable but is "sub-optimal", that is it does not always find the optimal solution. The proposed heuristics are as follows:

- We use the "bounded-queue" heuristic (discussed in section 6.5) which limits the number of tree-paths in the queue $Q\mathcal{G}_i^p$ to a fix number "b". While generally this restriction does not necessarily exclude the optimal solution, it reduces the exponential time and space complexity of maintaining and sorting the queue of tree-paths to time and space complexity $O(b)$, where "b" is the maximum size of the bounded-queue.

- 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 quantity $n = |N^s|$ is reduced into $s = |N^{sr}_{g(i)}|$ nodes. Considering the time complexity of the proposed multi-phase search process using BQ-A^*, i.e., $O(qps^p(h + qsp + b))$, the search space reduction will relax the complexity of the search algorithm to $O(qp(\frac{n}{10})^p\ (h + q(\frac{n}{10})p + b))$, assuming that each source-region is a 10'th of the whole search-space.

Chapter 8

Case studies

8.1 Objectives and categories of case-studies

This Chapter presents a set of experimentations related to the time complexity, space complexity, and accuracy of the architecture recovery technique in this thesis. The proposed technique has been implemented in Alborz [87], a prototype toolkit that aims to recover the architecture of medium size systems implemented in a procedural language such as C. The limitations of the proposed technique have been discussed in Section 1.8 of the thesis. Basili and Selby [18] have proposed four paradigms for experimentation and empirical studies in software engineering that are meant as guidelines for setting up and conducting experimentation. These paradigms consist of: i) *improvement paradigm*, ii) *goal-question-metric paradigm*, iii) *experimentation framework paradigm*, and iv) *classification paradigm*. In this respect, the case studies in this Chapter belong to the "experimentation framework paradigm" that consists of four categories corresponding to the phases of the experimentation process, as: “definition”, “planning”, “operation”, and “interpretation”. The *definitions* of the group of experimentations in this Chapter are presented below and the *operations* and *interpretations* of the experimentations are presented in the corresponding Sections.

The experimentations in this Chapter have been organized to:

1. Demonstrate the generality of the proposed approach by experimenting with systems in different domains such as expert systems, operating systems, distributed systems, and monolithic applications.

2. Evaluate the off-line time and space complexity of the architectural recovery technique as a function of source-code size, in Section 8.4.

3. Evaluate the on-line time and space complexity of the architectural recovery based on the size of the recovered components, in Sections 8.5.1 and 8.5.2.

4. Evaluate the usefulness of the approach in terms of stability, quality, and accuracy of the proposed pattern-matching process, in Sections 8.5.3, 8.5.4, and 8.5.5.

5. Demonstrate the user involvement and the incorporation of domain/system knowledge in the architecture recovery process, in Section 8.6.

8.1.1 Experimentation suite

The experimentations are performed on six middle-size industrial systems, namely: i) *Xfig* drawing editor [3], ii) *Clips* expert system builder [4], iii) *Bash* Unix shell

	1	2	3	4	5
System	*Source KLOC*	*No. of files*	*No. of functions*	*No. of Aggr. types*	*No. of Global vars*
Xfig 3.2.3	74	98	1662	37	1356
Clips 4.20	40	44	736	54	161
Apache 1.2.4	38	42	709	42	95
Bash 2.03	44	47	1017	45	365
Elm 2.5.6	35	62	420	19	244
GhostView 3.5.8	39	47	469	10	382

Table 8.1: Source-code statistics of the six case-study software systems. The presented data include: 1) size of the system in Kilo Lines Of Code (KLOC); 2) number of system files; 3) to 5) numbers of system's functions, aggregate datatype, and global variables, as defined in the abstract domain model in Section 3.1.1.

[5]; iv) *Apache* http server [6]; v) *Elm* Unix mail system [8]; and vi) *Ghostview* postscript file viewer and navigator [9]. Table 8.1 presents the source-code related characteristics of the experimentation suite.

8.1.2 Experimentation hardware platform

The hardware platform for the experimentation consists of a Sun Ultra 10 with 440MHZ CPU, 256M memory, and 512M swap disk. The experimentations are performed in a single-user load environment.

8.2 Alborz: software reverse-engineering toolkit

As a part of this work, the proposed pattern matching approach to architectural recovery has been implemented in a toolkit (Alborz) [87]. The toolkit offers an interactive environment for recovering and evaluating the architecture of a software system in terms of high-cohesive components. In addition to the pattern matching technique presented in this thesis, the Alborz toolkit provides: i) a partitioning clustering as presented in [91], ii) an incremental optimization clustering as presented in [89], and iii) an evaluation of the recovered software design views as presented in [88]. The tool has been built using the Software Refinery environment (Refine)[1] [85], Refine's built-in parser to parse the software systems written in C, and the built-in parser generator to design a parser for the Architecture Query Language (AQL).

The pattern-matching environment provided by the Alborz tool is illustrated in Figure 8.1 and consists of two phases, the off-line phase and the on-line phase that are discussed below.

8.2.1 Off-line pre-process phase

In the pre-process phase, illustrated on the left part of Figure 8.1, the source-code information is extracted from the software system. This information is processed and stored in a database in order to be used for the on-line analysis phase. The stored information is general enough to allow a programming language independent software architecture analysis. Currently, the software system is either parsed into an abstract syntax tree AST (using the Refine parser for C language) or parsed into an entity-relationship format (namely RSF) using the Rigi parser [1]. Based on the abstract domain model defined in Chapter 3 the parsed software system is represented as an attributed relational graph suitable for architectural analysis. Using

[1]The current version of Alborz consists of 30 KLOC written in the Refine's language.

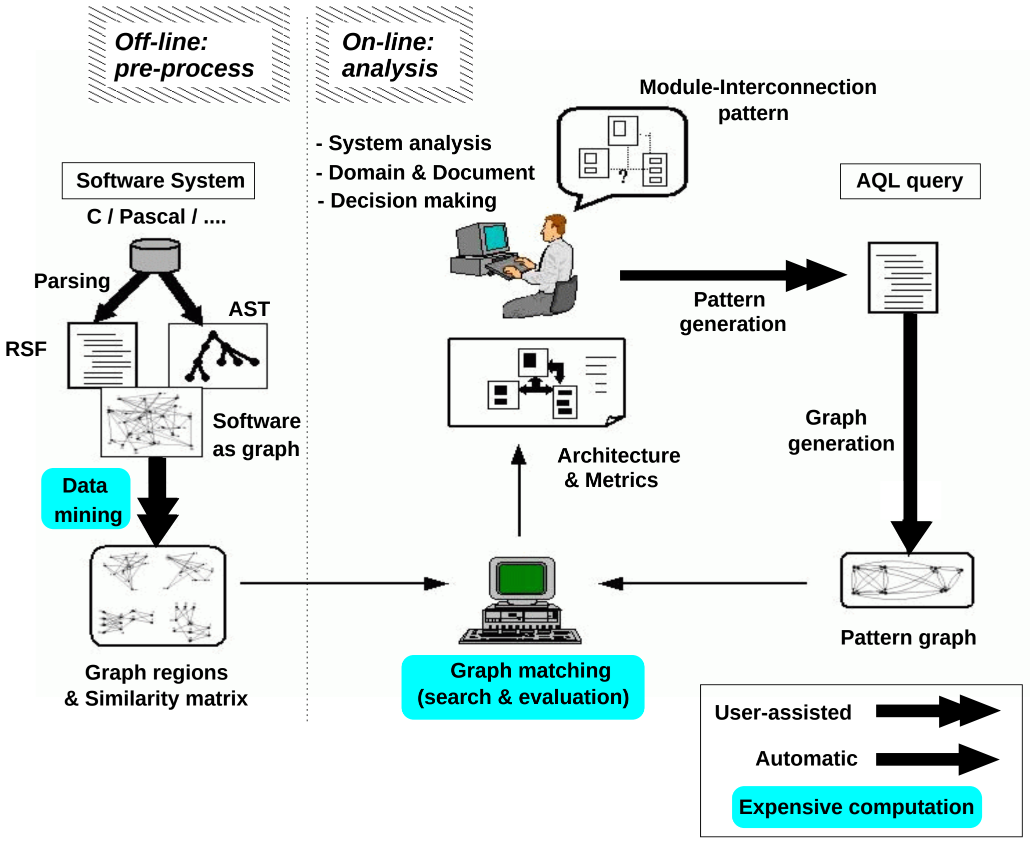


Figure 8.1: The interactive environment for the proposed pattern-based software architecture recovery.

data mining techniques the graph of the system is divided into graph regions (i.e., source-regions G_j^{sr}) which are then stored in a database. The off-line analysis can be a very time consuming process depending on the number of the system entities and the level of inter-relationship among them, however the off-line operations are performed only once for each system.

8.2.2 On-line analysis

In the right part of Figure 8.1(a), using the Alborz tool system analysis [89, 91], domain knowledge, and/or system documents, the user develops a hypothesis about the architecture of the system (i.e., conceptual architecture) that can be represented as a "module interconnection" pattern using the Architecture Query Language (AQL) discussed in Chapter 5. The minimum/maximum sizes and types of both the modules (subsystems) and their interconnections are considered as parameters to be decided by the user. An AQL query represents a pattern for a part or the whole system architecture to be recovered. The AQL query is parsed to generate a pattern-graph which is the expanded form of the AQL query. A pattern-graph consists of smaller patterns one for each module that are connected through groups of edges, where hard and soft constraints control the number of node/edge insertions/deletions in the matching process.

The pattern-graph and the selected graph regions from the database are supplied to the graph-matching engine that computes a sub-optimal match between the two graphs. The graph-matching engine performs approximate matching by minimizing an association-based cost function in an optimization search algorithm. The result of matching is presented to the user through easily understood information in HTML pages or graphs via graph visualization tools. The user investigates the results, and if needed, he/she expands the pattern in the AQL query and/or adjusts the interaction constraints between the components and repeats the matching process.

In the proposed pattern-based architectural recovery the pattern is incrementally generated using: association relation among the system entities [88]; clustering techniques [89]; domain related reference architectures; available system architecture documents; or consultation with the system developers. The objective in any of these methods is to extract small groups of system entities which represent the core functionality of the modules (or subsystems) in the system architecture. These groups are used to incrementally generate a constrained graph of modules and interconnection links represented by an AQL query. The steps for pattern generation are discussed in Section 8.7.1.

The Alborz toolkit also provides metrics for the evaluation of the software system and its architecture based on inter/intra-component association [88] and inter/intra-component connectivity [72].

8.3 Tractability of the recovery process

The proposed environment for architectural recovery in Figure 8.1 incorporates several techniques in order to tackle the inherent complexity of the architectural recovery process and to provide a tractable and interactive recovery environment. These techniques are enumerated below:

1. **Increasing minimum-support in data mining:** in Section 8.4 the techniques for dealing with expensive computations of extracting data mining frequent itemsets are discussed. The frequent itemsets are the basis for generating smaller search spaces and association-based similarity metrics to produce cohesive components.

2. **Sub-optimality to achieve performance**: in Section 6.5 the algorithm Bounded-path Queue A^* (BQ-A^*) is defined to reduce the space complexity of the search algorithm at the expense of obtaining possibly a sub-optimal solution. The on-line experimentations in Section 8.5 discusses this issue.

3. **Search space reduction:** in Section 7.4 the effect of reducing the whole search space into smaller search spaces on relaxing the complexity of the search algorithm BQ-A^* is discussed.

4. **Implementation considerations:** in Section 7.3.1 the efficient implementations of the connector-edges and edge-bundles are discussed that are crucial in reducing the complexity of the matching process.

5. **Hierarchical recovery:** the proposed recovery environment in Figure 8.1 allows to perform architectural recovery at two levels of granularity for the system entities, as "file-level" and "function-level", that contributes towards limiting the complexity of the recovery of modules of simple entities within a large software system.

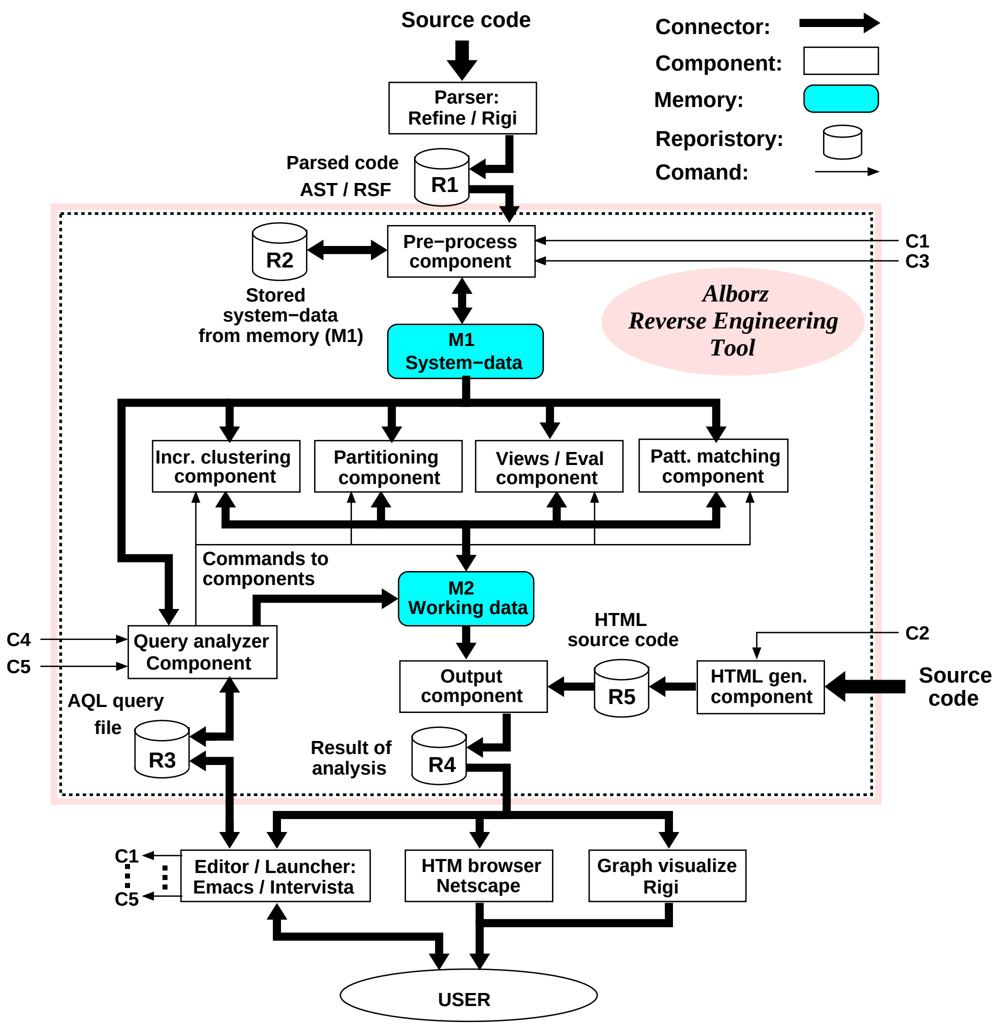
8.3.1 Architectural design of Alborz toolkit

The architectural design of the Alborz tool consists of a number of components with simple and well-defined interfaces and with a *pipe and filter* architectural style. Each component (filter) processes its input data in the form of a repository file or a memory data-structure (pipe) and stores the results in another file or data structure for the next component. Figure 8.2 illustrates the architecture of the Alborz tool (inside a dashed box) and its interaction with the surrounding environment. The contents of the repositories and memory units as well as key interface commands are shown in Figure 8.2.

Toolkit environment: the interactive environment of the Alborz tool is shown outside the dashed box in Figure 8.2, and consists of: i) the repository R1 to store the parsed software system as AST artifacts generated by the Refine's parser, or RSF tuples generated by the Rigi parser; ii) *Intervista*, a GUI tool in the Software Refinery environment, for formulating the AQL query and launching the commands; iii) the Gnu Emacs editor for developing and debugging the Alborz tool; iv) a Web browser such as Netscape for viewing and navigating the source-code, the results of analysis, and the tool generated metrics; and v) the Rigi tool for visualizing the recovered architecture of the system where, the boxes are the analyzed components and the arrows are either the resource interaction (i.e., import/export) between the components or their association strengths.

Toolkit components

- *Pre-process component*: during the off-line analysis this component is responsible for: i) extracting the system's entity-relationship data from the AST or RSF format of the parsed software system and storing it in the repository R1; ii) generating the system data such as frequent-itemsets, entity domains, and similarity matrix for the system entities; and iii) storing the entity-relationships and system data in the repository R2. The products of this phase have been discussed in Chapter 3. During the on-line analysis this component restores all the stored data in repository R2 into the memory M1



Memory		Repository	Command	
M1 – Entities & relations – Baskets of items – Frequent itemsets – Domains of entities – Similarity matrix	**M2** – Search tree of graphs – Recovered components – Association views – Partitioned clusters – Metrics	**R1** (input) – Refine AST objects – Rigi RSF tuples	C1: Generate system data and store them in R2 C2: Annotate source files and store them in R5	Off−line
		R4 (output) – Netscape HTML pages – Rigi RSF files – Emacs Text files	C3: Inititialize tool C4: Generate query C5: Analyze query	On−line

Figure 8.2: The architectural design of the Alborz reverse-engineering tool.

to be used by the on-line analysis.

- *Analysis components*: the major tasks of the tool in the on-line analysis are performed by four analysis components in Figure 8.2. The operations of the pattern matching component are the subject of this thesis, however the operations of the other components, i.e., incremental clustering [89], partitioning clustering [91], and software design views and evaluation [88], are not presented in this thesis as these are independent applications.

- *Query analyzer component*: this component performs two tasks: i) generates a template AQL query including the tool-suggested main-seeds, default sizes for abstract-components, and unconstrained sizes for abstract-connectors in order to allow the user to tailor the sizes and types and proceed with the on-line analysis; and ii) parses the query file using the AQL grammar and domain-model; checks the semantics of the query information; and dispatches the activation commands to an appropriate component according to the directives in the header of AQL query.

- *Output component*: generates both the HTML pages and the graph representation for the recovered architecture, design views, and the computed metrics.

- *HTML generator component*: produces HTML source-code by annotating the system's source-code files.

	1	2	3	4	5	6	7	8
System	*source code* (KLOC)	*# nodes*	*# edges*	*AST in disk/mem.* (MB)	*stored node/edge* (KB)	*stored FQI* (KB)	*stored Domains* (KB)	*Stored Matrix* (KB)
Xfig	74	3055	16387	54 / 232	90 / 129	353	4800	5500
Clips	40	951	3584	9.1 / 30	25 / 28	61	478	556
Apache	38	846	2176	14.5 / 47	24 / 19	34	155	183
Bash	44	1427	4562	20.5 / 71	45 / 37	71	358	421
Elm	35	683	2766	11.2 / 37	19 / 21	120	383	441
GV	39	861	1970	36.5 / 146	26 / 17	21	80	102

Table 8.2: Off-line space utilization statistics of the six experimented software systems.

8.4 Off-line experimentations

In this Section, the *off-line* phase experimentations are presented. The groups of frequent itemsets, entity domains, and the similarity matrix constitute the major input data for the pattern-matching process. These data are generated and stored once and are used several times.

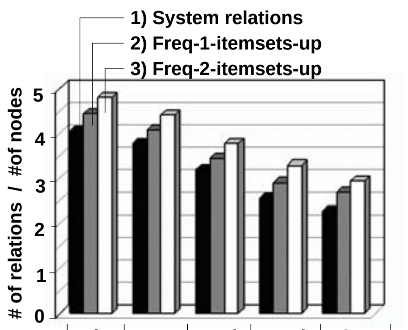
8.4.1 Off-line space complexity

In the off-line analysis, the main effort focuses on generating frequent itemsets that include all the relations *use-F, use-T, use-V* having the lowest possible minimum-support value, i.e, 2. Unfortunately, the number of intermediate frequent itemsets is very sensitive to the chosen minimum-support value and for small minimum-support values the number of frequent itemsets increases very rapidly hence they require a large swap disk and execution time.

In Table 8.2 the required space for the six case study systems at the "function-level" are shown. These measures pertain to major system data that are contained in memory M1 and repository R2 in Figure 8.2. Columns 2 and 3 show the size of entities and relationships that constitute the source-graph $G^s = (N^s, R^s)$. The entities (nodes) are of different types *Function-abs, Type-abs*, and *Variable-abs* whose details are shown in Table 8.1; and the relations (edges) are of different types *use-F, use-T*, and *use-V*. The last five columns present the disk or memory space requirements for storing the different system data to be used for on-line analysis. Column 4 presents both the memory and disk space utilization of the system's abstract syntax tree (AST) that is generated by the Refine's parser as they are related to the source-code size. Column 5 presents the required disk space for storing the system entities and relationships (quantities are in columns 2 and 3) that are extracted from the corresponding AST.

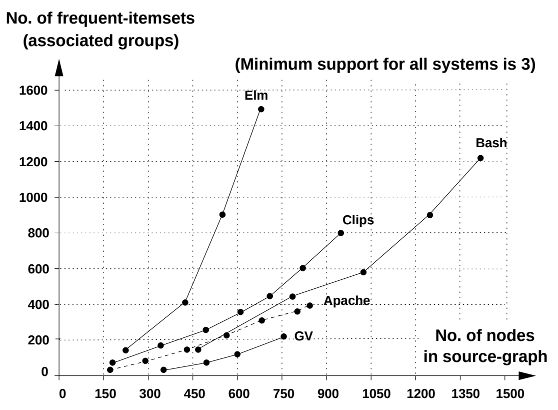
By storing the extracted entity-relationships on disk a large amount of memory space is saved during the on-line analysis. On average the space reduction ratio obtained by storing/restoring entity-relationships on disk as opposed to uploading or computing the AST is $\frac{AST\ disk\ space}{entity-relation\ disk\ space} = 303$. Therefore, it is necessary to compute and store the entity relationships in the pre-process phase. Moreover, the size of AST in memory is very large and the size increase ratio from file to memory is $\frac{AST\ memory\ space}{AST\ disk\ space} = 3.86$. The large size of AST in memory (e.g., 232 MBytes for Xfig) is one of the major obstacles in analyzing large systems at once. Columns 6 to 8 present the sizes of the stored frequent-itemsets, entity domains, and similarly matrix on disk, respectively.

Figure 8.3(a) illustrates the ratio between the total number of relations of types

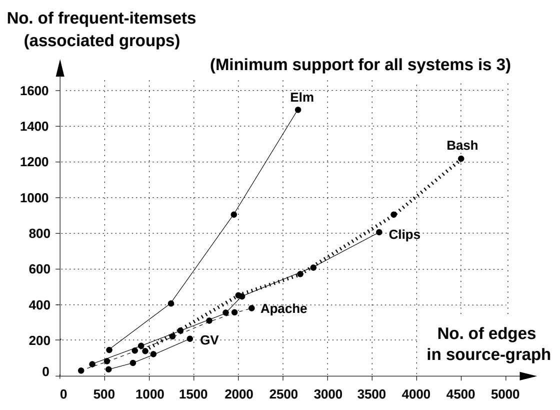


	Elm	Clips	Bash	Apache	GV
# of entities	683	951	1427	846	861
# of relations	2766	3584	4562	2176	1970
Systems before data mining process					

(a)



(b)



(c)

Figure 8.3: (a) The relation-to-node ratio: 1) in the case studies before data mining; 2) in the generated frequent-1-itemsets and higher; and 3) in the frequent-2-itemsets and higher. The ratio increases in each subsequent frequent-itemsets. (b) and (c) The number of generated frequent-itemsets versus the number of nodes (or relations) in the studied software systems. Systems with higher relation-to-node ratio in part (a) generate more associated groups with a given number of nodes.

use-F, use-T, use-V to the total number of nodes of types *Function-abs, Type-abs, Variable-abs* in each studied software system. This ratio is an indication of the overall data/control-flow complexity of a system. In the group of bars labeled "1) system relations" the highest ratio belongs to Elm system with the $\frac{\#relations}{\#nodes} = 4$ and the lowest belongs to Ghostview with ratio 2. The application of Apriori algorithm generates the frequent-itemsets of entities from frequent-1-itemsets to frequent-k-itemsets where k is the maximum size of the extracted itemsets. As it is seen in Figure 8.3(a):

$$\textit{ratio for frequent-2-itemsets} > \textit{ratio for frequent-1-itemsets} > \textit{ratio for original system}$$

However, the number of associated groups decrease for a higher itemset size. Since the associated groups in frequent-1-itemsets have only one shared entity in common, it makes sense to consider frequent-1-itemsets as noise and delete them, hence consider the associated groups in frequent-2-itemsets and up (i.e., two, three, etc. entities in common). This causes to compute the similarity between system files based on only large associated groups of entities, hence recovering more related components.

Figure 8.3(b) illustrates a comparison of the tested systems in terms of the number of generated associated groups versus the number of entities in the systems, all having the relations *use-F, use-T, use-V* and minimum-support 3. The following observations can be made from the curve of each system in Figure 8.3(b): i) the rate of generating associated groups is increasing with respect to the size of the entities, where this increase is caused by forming new associated groups whose entities are partly in the newly added entities and partly in the previous entities; ii) systems with higher relation-to-node ratio in Figure 8.3(a) generate more associated groups with a given number of nodes, e.g., Elm has higher relation-to-node ratio than Clips and the curve of Elm is above of the curve of Clips in Figure 8.3(b); and iii) the number of the generated groups are kept within a tractable size by this increase.

Figure 8.3(c) provides complementary information to the above discussion, where the number of generated associated groups versus the total number of relations *use-F, use-T, use-V* and minimum-support 3, are shown. An interesting observation in this case is that almost all the curves (except for the Elm system) mostly overlap as shown in Figure 8.3(c). The comparison of the Figures 8.3(b) and (c) suggests that: i) the number of generated associated groups is almost proportional to the number of a system's relations and not the number of entities; and ii) the number of edges in a system is a good indication of the number of frequent-itemsets in a particular minimum support.

Figures 8.4(a) and (b) illustrate the application of the Apriori algorithm by considering the number of the generated intermediate frequent-itemsets versus the size of itemsets, for different systems. The data mining Apriori algorithm [12] is an iterative process, where the *frequent k-itemsets* are obtained by k passes over the database of baskets and items. In the first pass, the algorithm only counts the occurrence of each item in all baskets to determine the *frequent 1-itemsets.* In each subsequent pass, e.g., pass i, the frequent i-itemsets are computed from the frequent 'i-1'-itemsets obtained in the previous pass. In Figure 8.4(a) all the intermediate frequent-itemsets are shown. In the beginning the number of generated frequent-itemsets increases rapidly by the size of the generated itemsets. As Figure 8.4(a)

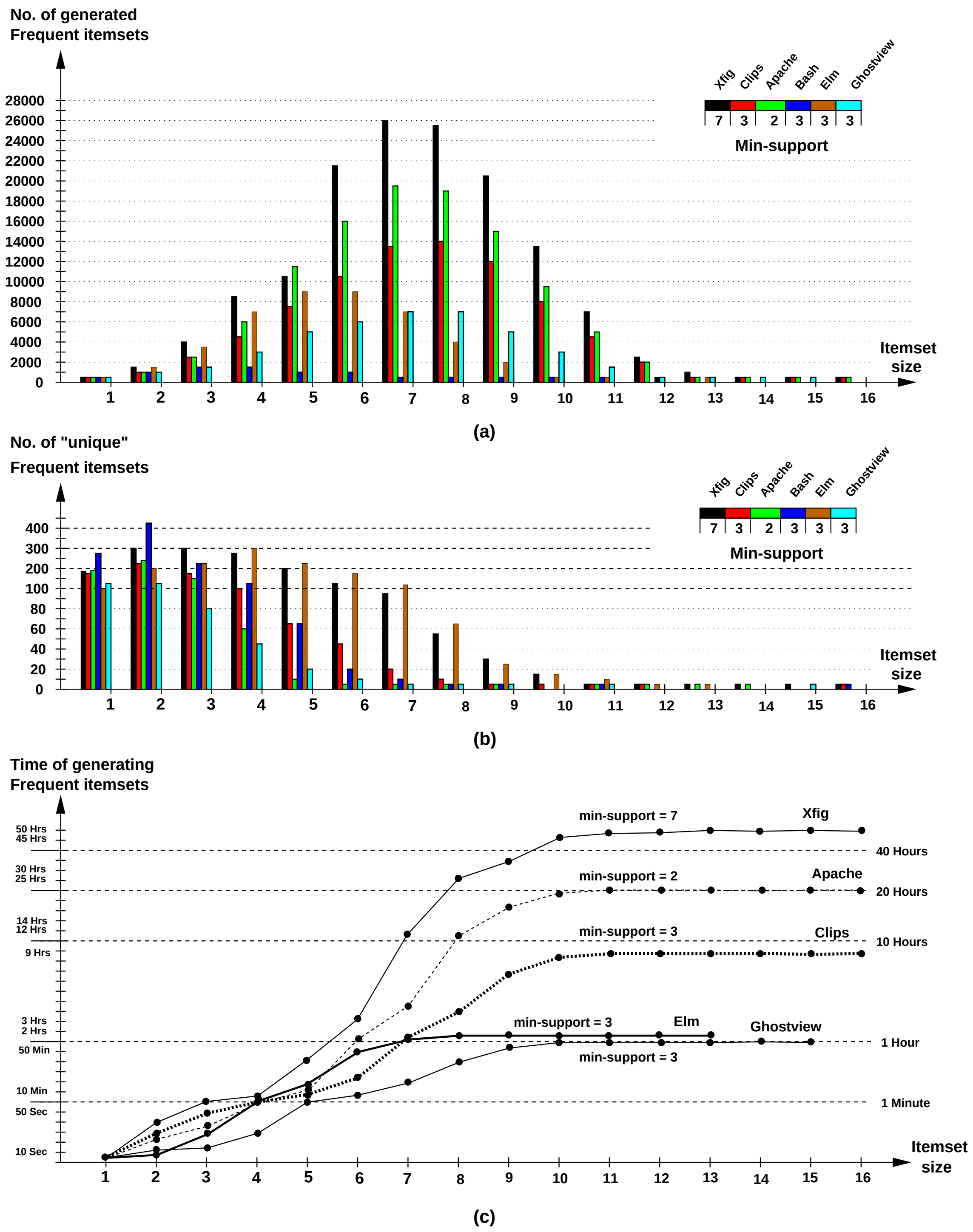


Figure 8.4: Quantity and time characteristics for generating frequent itemsets versus size of itemsets for different systems. (a) All the generated intermediate frequent itemsets are considered. (b) The unique frequent itemsets extracted from (a) are shown. (c) Time of generating frequent itemsets in (a).

shows, the number of generating frequent-itemsets highly increases for the itemset sizes 4 to 8. However, after passing the maximum size that occurs around itemset size 7 and 8 the numbers decrease with almost the same speed that they increased. However the number of final frequent-itemsets is small and in all studied systems it is less than 5.

Figure 8.4(b) illustrates the number of frequent-itemsets at each itemset size, where no associated group is a subset of any other associated-group. In other words these associated-groups are "unique". The unique associated-groups are used to compute the entity association "*entAssoc*" similarity metric defined in Section 3.3, and consequently generate the group of entity-domains and similarity matrix for cost evaluation in the pattern-matching process.

As opposed to the highly-increasing number of the intermediate frequent-itemsets in Figure 8.4(a), the number of the unique frequent-itemsets in Figure 8.4(b) are monotically decreasing from frequent-2-itemsets. As it will be discussed later, the number of intermediate frequent-itemsets are very sensitive to the value of minimum-support which allows the user to control the space and time complexity of the pre-process phase.

8.4.2 Off-line time complexity

The experimentation in Figure 8.5 illustrates the time of generating the frequent-itemsets versus the minimum support value for the studied systems. The itemset size of the final frequent-itemsets is shown at the top of each bar.

This experimentation signifies the process of obtaining a feasible minimum-

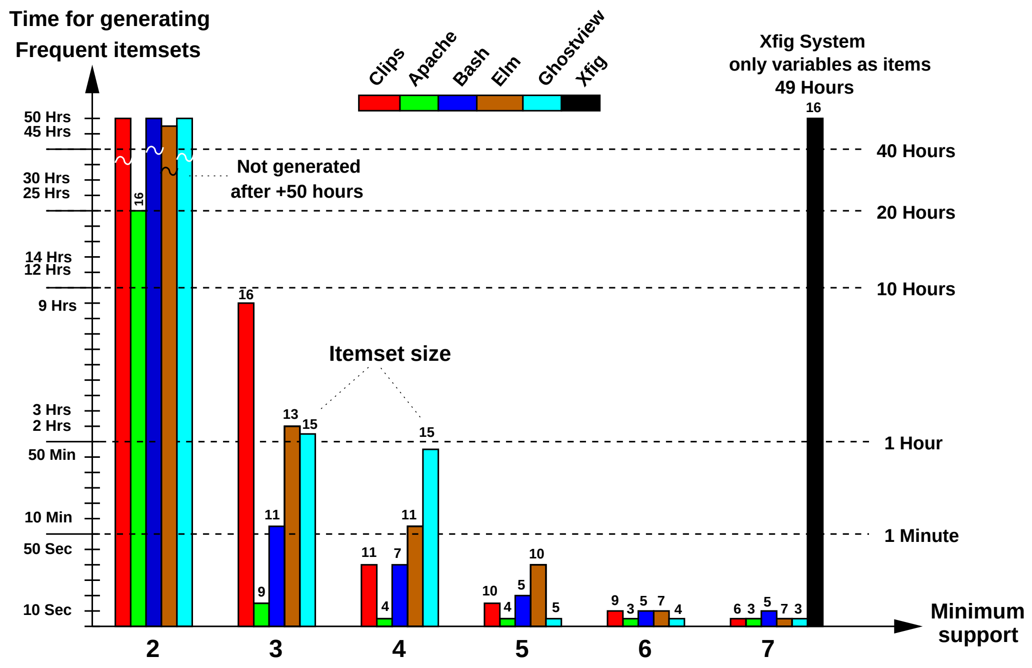


Figure 8.5: Time of generating frequent itemsets versus minimum support value for the studied systems. Except for one system the number of frequent itemsets with minimum support 2 explodes and after 50 hours no result is obtained.

support value to extract the frequent-itemsets for a system. Except for the Apache system that yields the set of frequent-itemsets with minimum support 2 in 20 hours, the other systems failed to produce any result after 50 hours and more. Therefore, the user would increase the minimum-support value to 3 and run the Apriori algorithm again. With this value, four other systems except the Xfig system produce frequent-itemsets that are stored in the repositories for further analysis. In order to obtain a result for Xfig system, the user may increase the minimum-support value to 4 and higher until the Xfig system yields the frequent-itemsets at minimum-support 7 in 49 hours. At it is seen in Figure 8.5 the generation time for the frequent-itemsets decreases very fast with the increase of minimum-support value.

	1	2	3			4
System	*source code* (KLOC)	*parsing time* (Min)	*Frequent itemsets* *Min-sup* [*Max-items*]	*quantity*	*time* (Hr : Min)	*Similarity Matrix* (Sec)
Xfig	74	45	7 [16]	3167	49 : 18	274
Clips	40	3	3 [16]	810	8 : 53	11
Apache	38	8	2 [16]	678	20 : 32	12
Bash	44	5	3 [11]	1225	0 : 03	13
			2 [max 4]	18926	0 : 46	
Elm	35	10	3 [13]	1525	1 : 42	28
GV	39	27	3 [15]	411	1 : 04	6

Table 8.3: Off-line time statistics of the six studied software systems.

Therefore, the minimum-support value can be considered as a control mechanism to perform a tractable computation of the Apriori algorithm in generating the frequent itemsets.

In Figure 8.4(c) the time of generating frequent-itemsets versus the itemset size during the Apriori algorithm operation with an optimum minimum-support value for each system (discussed above) is shown. The required time highly increases for the itemset sizes between 5 to 9. This property suggests to run the Apriori algorithm with a small minimum-support value, and stop the algorithm before the itemset size 5 to obtain more association relations between entities. We will use such a property later.

In Table 8.3 the time statistics of the generated frequent-itemsets for the selected systems are presented. For the Xfig system, even though the minimum-support threshold has been increased to 7 still the maximum size of the generated itemsets is 16 that means large associated groups have been extracted. The combination of the maximum-itemset size and the number of extracted associated group, (i.e., 3167 for Xfig), is a criterion for the user to assess the quality of the generated associated groups. Ideally, we would like to generate the frequent itemsets with minimum-support 2 to take into account all the associated groups. However, the explosive increase in the number of associated groups prevents such a computation. In such cases, still it is possible to obtain enough relations among the system entities by multiple application of the Apriori algorithm with different minimum-support values, as in case of the Bash system in Table 8.3. For the Bash system, the minimum-support 3 produces 1225 associated groups with maximum itemset size 11. However to increase the number of associated groups of entities, the algorithm is applied again with minimum-support 2, but it is forced to terminate after generating frequent-4-itemsets, that is before the number of associated groups becomes very high. In this case, the resulting associated groups produce entity association similarity measures among those entities that did not exist in the previous run of the algorithm with minimum-support 3. Since the algorithm is prematurely stopped, the recovered association values are probably lower than the actually values. In the case of the Clips system, the minimum-support is 3 which produces 810 frequent itemsets with different itemset sizes and max-itemset size of 16. This group of frequent-itemsets are enough to produce an accurate analysis result.

8.5 On-line experimentations

In this Section, the different aspects of the pattern matching search engine such as time/space complexity, stability, accuracy, and quality are presented through a series of experimentations.

8.5.1 On-line time complexity

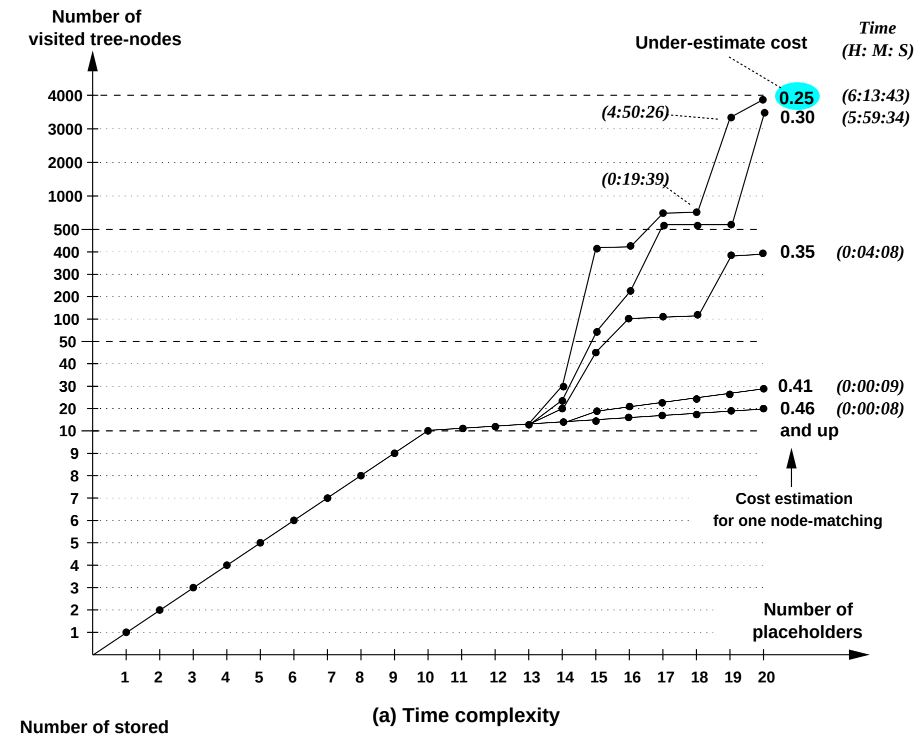
Figure 8.6(a) illustrates the time complexity of the pattern-matching algorithm versus the number of placeholder-nodes at function-level and for one module. In this case, the estimation of the remaining cost for matching a system entity with a placeholder is used as parameter. The under-estimate cost "$cost_{uest}$"[2] is based on the assumption that all the remaining nodes to be matched from the current tree-node down to a leaf tree-node have maximum similarity values to the main-seed.

However, to limit the complexity of the A^* algorithm we use overestimate costs and the bounded-queue heuristic at the expense of obtaining a sub-optimal solution. An over-estimate cost is the cost of matching a node with a placeholder such that the similarity of the node with the main-seed is less than the maximum similarity and more than average similarity to the main-seed. The computation time for each curve is shown beside each cost.

Moreover, as discussed in Section 7.4 the following trade-offs have already been made which reduce the complexity of the A^* search to produce sub-optimal results: i) decomposition of the search-space into domains; and ii) using bounded path-queue for incomplete tree-paths. Without such heuristic trade-offs the search time and space complexity would be intractable even for a small system.

In Figure 8.6(a), for either the under-estimate cost or an over-estimate cost the *BQ-A^** search first collects all the entities of an "associated group" with maximum

[2] In this experimentation the similarity metrics are normalized, hence under-estimate cost is 0.25.



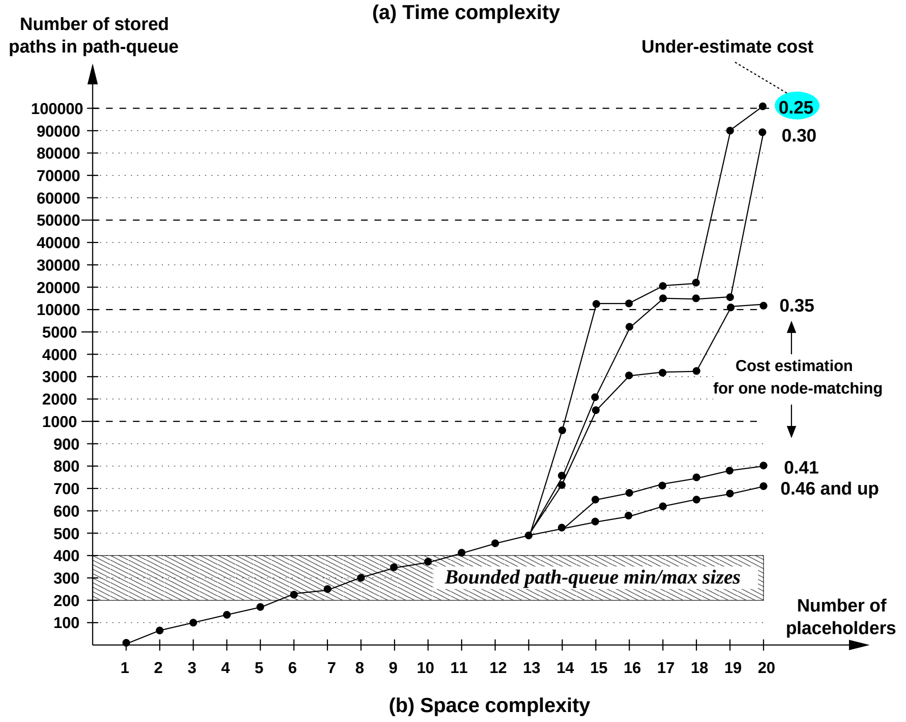


Figure 8.6: (a) Time complexity of the matching algorithm in term of the visited tree-nodes. (b) Space complexity of matching algorithm in term of the stored paths in the search path-queue.

in-between similarity values (*entAssoc*). This case was shown in detail in the example of Figure 6.11 in page 150, where four entities with the in-between $entAssoc = 4$ were selected. After selecting all the entities in the associated-group with maximum *entAssoc* the search proceeds with selecting further entities from other associated groups which produce a minimum total cost ($cost_{total} = cost_{path} + cost_{uest}$).

8.5.2 On-line space complexity

Figure 8.6(b) illustrates the space complexity of the pattern-matching algorithm in terms of the number of stored paths in the path-queue $Q\mathcal{G}_i^p$ versus the number of placeholders at function-level and for one module.

In this experimentation, the same parameters as in the time complexity experimentation are used to obtain space utilization results. In the case of the underestimate cost, the size of the path-queue is rapidly increasing. The original A^* search keeps all the evaluated tree-paths in the path-queue, therefore the space explosion usually occurs long before the time explosion occurs. The trade-off between optimality and performance discussed in Section 7.4 can save a lot in both space and time performance of the searching process. At the bottom of Figure 8.6(b) a shaded area highlights the restricted sizes for path-queue when the bounded-queue heuristic, discussed in section 6.5, is used. In this case the number of stored paths in the bounded path-queue oscillates between the minimum and maximum sizes of the bounded path-queue as illustrated in Figure 6.9(c) in page 143.

8.5.3 Stability of the recovery

In this Section the stability of the search engine with respect to the change of the seeds in a module (or subsystem) is presented. The steps for stability checking at function-level are as follows:

- A subsystem is selected using the file-level analysis of the software system. This subsystem contains the related files that ensure their entities have high degree of interaction.
- Using the main-seed selection algorithm provided by the Alborz tool select a main-seed with large domain and generate a query with one module and a maximum size e.g., 20.
- From the file that contains the main-seed select 19 functions (or less, if 19 is not possible) and assign them to the module as the seeds of the query.
- In a series of 19 iterations do the following: i) delete the lowest similarity seed from the module; and ii) recover the module with the remaining seeds. At the last iteration all the seeds have been deleted and the recovery is based on solely the main-seed.

The stability of the recovery process is high if deleting the seeds has less affect on the result of the recovery. In other words, the recovery process is not distracted by the noise seeds and recovers the members of the same group of entities once a noise seed is deleted. Figures 8.7(a) to (c) illustrate the stability analysis of the search engine using three systems Clips, Apache, and Bash, respectively. For the

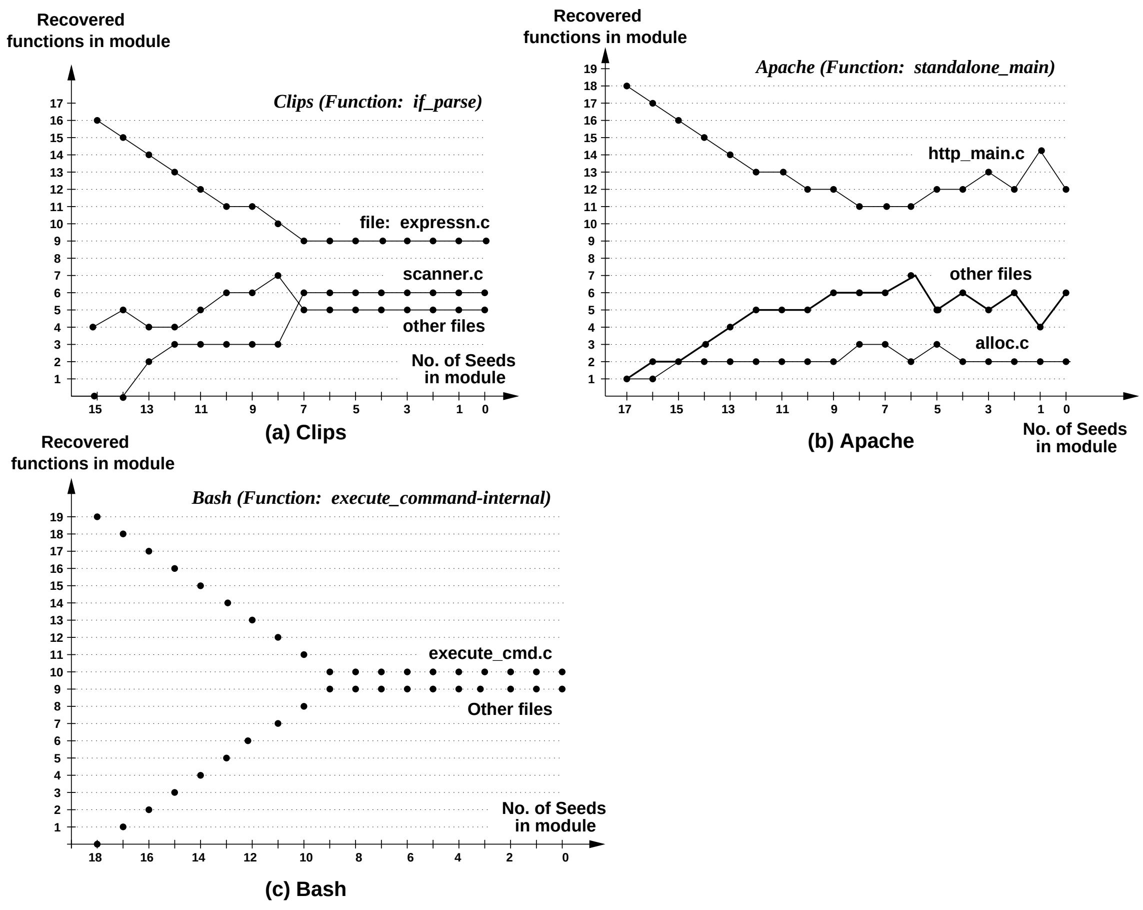


Figure 8.7: The stability of the searching algorithm.

Clips system, 15 seeds are selected from the file *expressn.c* and in a sequence of 15 recovery processes the seeds are deleted one-by-one. The first deleted seeds are replaced by functions from the file *scanner.c* and also by functions from *other files*. These newcomer functions constitute a stable group and most of them remain up to the end of the recovery. After deleting 8 seeds, the whole module becomes stable and each deleted seed is returned back to the module by the recovery process, up to the time that all seeds are deleted and only the main-seed is remained. In the other two cases for Apache and Bash in Figures 8.7(b) and (c), the same pattern as the case of Clips is repeated. However, for the Bash the pattern is more smooth than Clips, and in the case of Apache the recovery of the *http_main.c* functions is stable around 12 functions by an error margin of 2 functions.

8.5.4 Quality of the recovery

Figure 8.8 illustrates an experimentation to assess the modularity quality of the recovery process on five software systems. For each system, the experimentation is performed in a sequence of five recovery processes. In the first experimentation, a query with one module is recovered that yields two subsystems (including the rest-of-system). In the second experimentation, a query for two subsystems is recovered that yields three subsystems, and so on. At each iteration the modularity quality is measured according to a metric from the relevant literature, as defined below:

In this metric [72], the modularity quality is measured in terms of intra-/inter-connectivity among the entities in a collection of subsystems that form an architecture for the software system. This metric is referred to as the connectivity modularity-quality metric and is denoted as MQ_{con}. This metric is defined below:

$$MQ_{con} = \frac{1}{k}\sum_{i=1}^{k}\frac{e_i}{N_i^2} \quad - \quad \frac{1}{\frac{k^2-k}{2}}\sum_{i,j=1}^{k}\frac{e_{i,j}}{2N_iN_j}$$

where, k is the number of subsystems, e_i ($e_{i,j}$) is the number of relations of types *use-F, use-T, use-V* among the functions, datatypes, and variables in a recovered subsystem S_i (among two recovered subsystems S_i, S_j); and N_i (N_j) is the number of entities in the subsystem S_i (S_j). The first term evaluates the average intra-

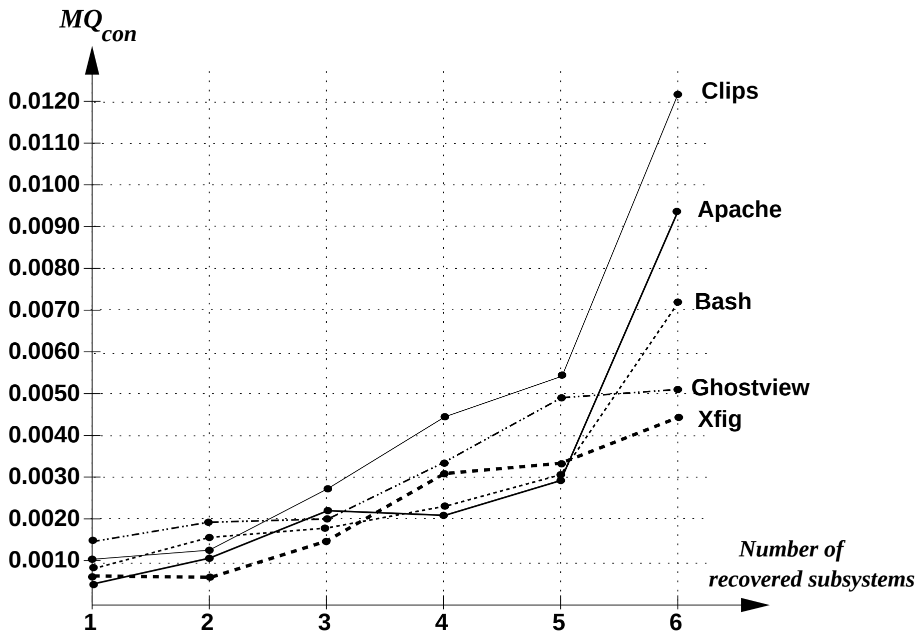


Figure 8.8: The modularity quality measure of five studied systems based on intra-/inter-connectivity among the recovered subsystems.

connectivity among entities in a subsystem S_i and the second term evaluates the average inter-connectivity among entities in every two subsystems S_i and S_j.

In Figure 8.8 the modularity quality of all five systems are increasing as more subsystems are recovered, which means the recovery process monotonically collects files with high degree of interaction to form subsystems (or modules).

8.5.5 Accuracy of the recovery

In this Section, the accuracy of the pattern matching process is demonstrated using the information retrieval *Precision* and *Recall* metrics [46]. These metrics are illustrated in Figure 8.9(a). In Figure 8.9(b) each node corresponds to the Pre-

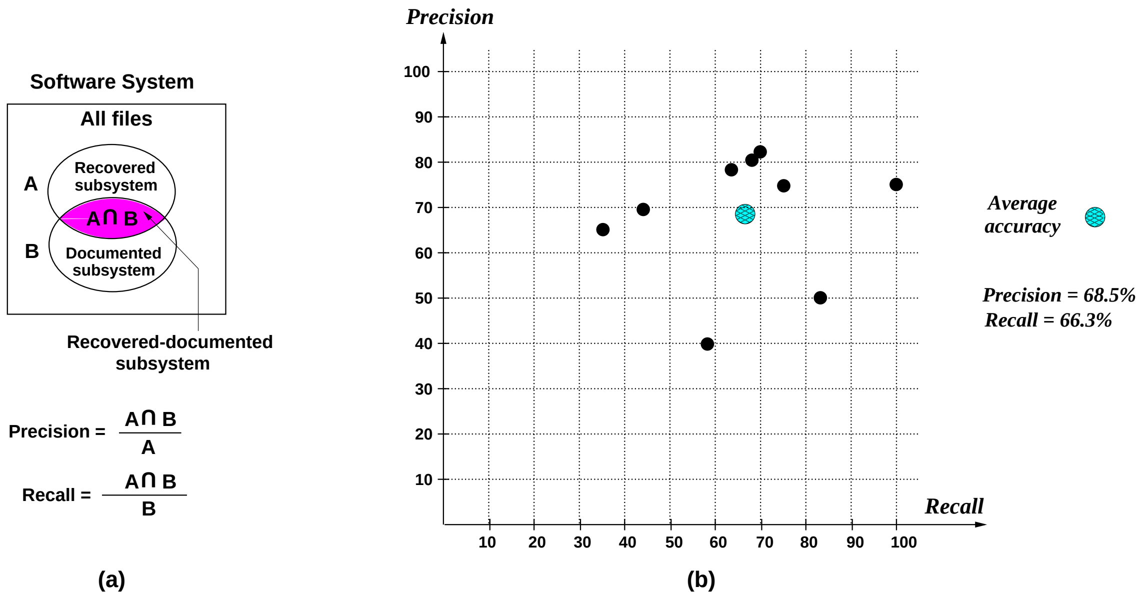
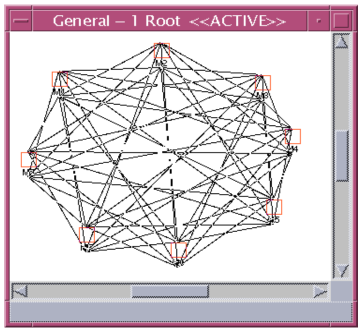


Figure 8.9: (a) Precision and Recall metrics are used as the measure of accuracy. (b) The accuracy of the pattern matching process and the average accuracy.

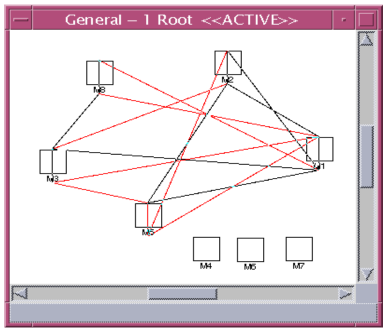
cision and Recall measure of a recovered subsystem in two case studies that will be discussed in Sections 8.7.2 and 8.7.3. The average accuracy, "Precision 68.5%" and "Recall 66.3%", is obtained by averaging the values of Precision and Recall for the nodes. The average accuracy represents the conformance of the recovered architecture with the documented subsystem structure of the two case studies and can be considered as a very promising accuracy.

8.6 User-assistance features

The proposed environment provides statistics and metrics at different stages of the recovery process to familiarize the engineer with the software system, and to assist in the recovery process. In this Section, the graph visualization and main-seed



(a) Import / export of entities among 8 modules



(b) Association among modules in part (a)

Figure 8.10: (a) Module-interconnection representation. (b) Module association representation.

selection features of the environment are discussed.

8.6.1 Graph visualization

The proposed environment provides means for visualizing the interaction among the software system files as well as the recovered modules/subsystems using i) import/export relation between entities, and ii) the association relation. Figures 8.10(a) illustrates the import/export interaction among 8 recovered modules of a software system, however, the import/export interaction is usually complex and difficult to interpret.

An alternative form of interaction visualization is through association relation among the components (i.e., files, modules, or subsystems). The association among two files is extracted from the *groupAssoc* similarity measure between two files (defined in Section 3.4), where the denominator of the formula for association is the size of one file instead of the total sizes of two files. As in the case of entity import/export, the number of associations among the files is also high. However, the

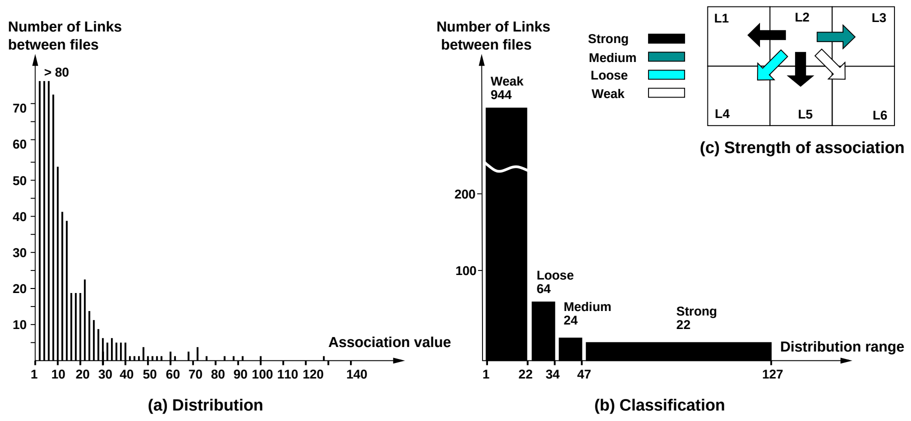
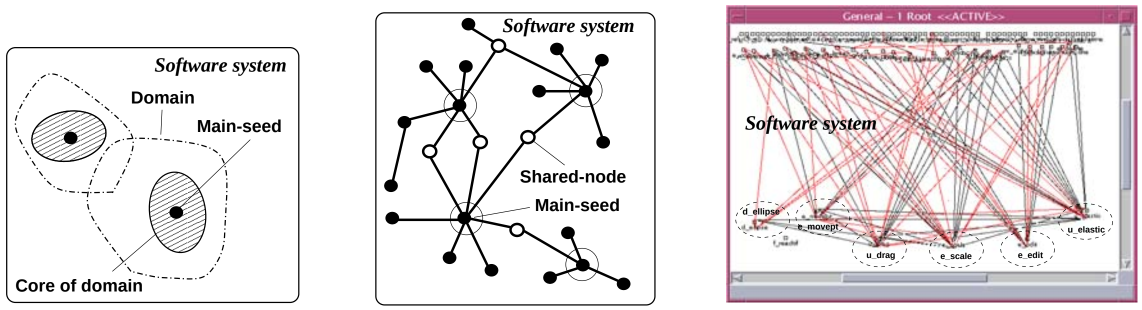


Figure 8.11: (a) Distribution and (b) Classification of the file association values in the Clips system. (c) Strength of association between file L2 and other files in a system of six files.

association values are distributed over a large range of values, hence they can be classified into several ranges, namely "strengths of association." This classification of values allows to simplify the visualization of the association graph of the system components (i.e., files, modules, or subsystems). A detailed discussion on generating the "component association graph" has been presented in [91]. Figures 8.11(a) and (b) sketch a typical distribution of the association values among the files in the Clips system and their classification into four ranges: *strong, medium, loose*, and *weak*. The group of edges in each class have been color-coded to be visualized or be selectively filtered out. The obtained graph of components and association links can be used to visualize and simplify the association structure of the system files. Figure 8.10(b) illustrates an example of an association graph with 8 modules which has been simplified to show only strong and medium association links.



(a) Domain analysis (b) Graph analysis (c) Graph visualization

Figure 8.12: Three methods provided by the proposed environment to generate AQL query main-seeds.

8.6.2 Main-seed selection

The challenging step in defining an architectural pattern corresponds to main-seed selection for the abstract components in the AQL query. The engineer may decide to use one or more of the following methods: 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. The adopted method(s) should be able to suggest important and rather distinct main-seeds as the cores of the abstract module/subsystems in the AQL query. The proposed environment provides three methods for main-seed selection as illustrated in Figure 8.12. The first two methods are automatic and the third method is manual, as discussed bellow.

Domain analysis (Figure 8.12(a)): uses a ranking mechanism to select the main-seeds whose domains are large, sufficiently distinct, and the entities in the core part of the domains have high level of association to the main-seed. This method is discussed in detail in [89].

Graph analysis (Figure 8.12(b)): uses the association graph of system entities and a ranking mechanism selects the main-seeds whose total association to other nodes is maximum. Where, the association value of a shared node decreases if it was already visited for selecting a previous main-seed. This technique is discussed in detail in [91].

Graph visualization (Figure 8.12(c)): uses the tool provided simplified association graph of the system files (as discussed earlier in this Section), where only the strong and medium association links are shown. The user can visualize the graph using a visualization tool such as Rigi [1] in order to drag the nodes around the screen and select the highly associated files as main-seeds.

The experimentations with the six software systems show a high degree of conformance among the obtained main-seeds by the above three methods.

8.7 Architectural recovery case studies

In this Section, two software systems Xfig and Clips are analyzed and the architecture of each system is recovered in terms of the collection of files using the steps for incremental pattern generation and architectural recovery as discussed below.

8.7.1 Incremental pattern generation and recovery

The architectural pattern defined in the AQL query is generated through an incremental and evolving pattern, as described in the following steps:

Step 1: Decide on a method of main-seeds selection for the AQL abstract modules/subsystems as discussed in Section 8.6.2. Initialize an AQL query with zero modules/subsystems.

Step 2: Select the main-seed for the next module/subsystem to create, and assign the number of placeholders, e.g., 10 for subsystem recovery and 20 for module recovery. Recover the new module/subsystem where no links 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[3]. 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 con-

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

strained 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.

8.7.2 Architecture recovery of Xfig

The source-code statistics of the Xfig system were presented in Section 8.1.1. According to personal communication with the maintainer of the Xfig system [99], Xfig 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. The system naming conventions in Xfig are as follows: d_* files relate to drawing shapes; e_* files relate to editing shapes; u_* files are utilities for drawing or editing shapes; f_* files have file-related proce-

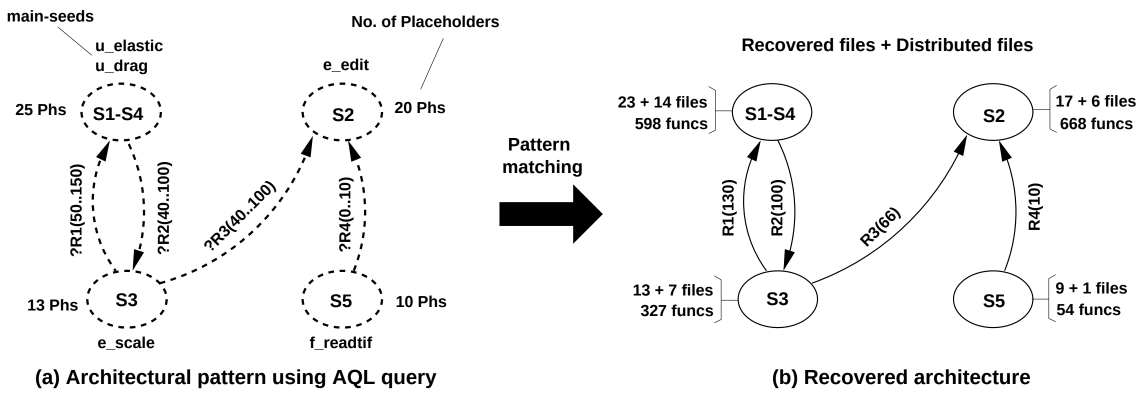


Figure 8.13: (a) The architectural pattern of the Xfig system where the subsystems S1 and S4 have been merged. (b) The recovered architecture where the link constraints have been satisfied.

dures; and *w_∗* files have X11 window calls in them to do all of the window-related functions.

Figure 8.13(a) illustrates the generated architectural pattern of the Xfig system with four abstract subsystems and corresponding link constraints. During the incremental and iterative recovery process this pattern yields the recovered architecture in Figure 8.13(b) where the size constraints for both the subsystems and links have been satisfied. The rationale for such a pattern is as follows. The initial observation from the file association graph of the Xfig system indicates that the files from editing, drawing, and utility subsystems are highly inter-related. This characteristic can be observed from the Xfig file association graphs in Figure 8.15(a) to (e) in page 217. Therefore, recovery of a well separated group of subsystems for Xfig is almost impossible. In this case, a proper strategy is to put these three subsystems into two subsystems with high-level of interaction, and separate them from the other subsystems of Xfig. With this objective the following pattern is generated. The tool provides the main-seeds according to the discussion of Section 8.6.2.

- The subsystems S1 and S4 collect the files from the inter-related subsystems editing, drawing, and utility subsystems. Therefore, during the iterative process two subsystems S1 and S4 have been merged into subsystem S1-S4, and the collection of domains for two main-seeds *u_elastic* and *u_drag* is used for the recovery of S1-S4.

- The subsystem S2 with main-seed *e_edit* recovers most of the windowing files, hence, it is independent of S1-S4 and no link constraints are needed between them.

- The subsystem S3 with main-seed *e_scale* also collects the files from utility and editing subsystems, hence the interaction between S3 and S1-S4 has been increased in both directions to encourage S3 to collect more files from the editing and utility subsystems to achieve our goal of generating two subsystems for editing, drawing and utility files.

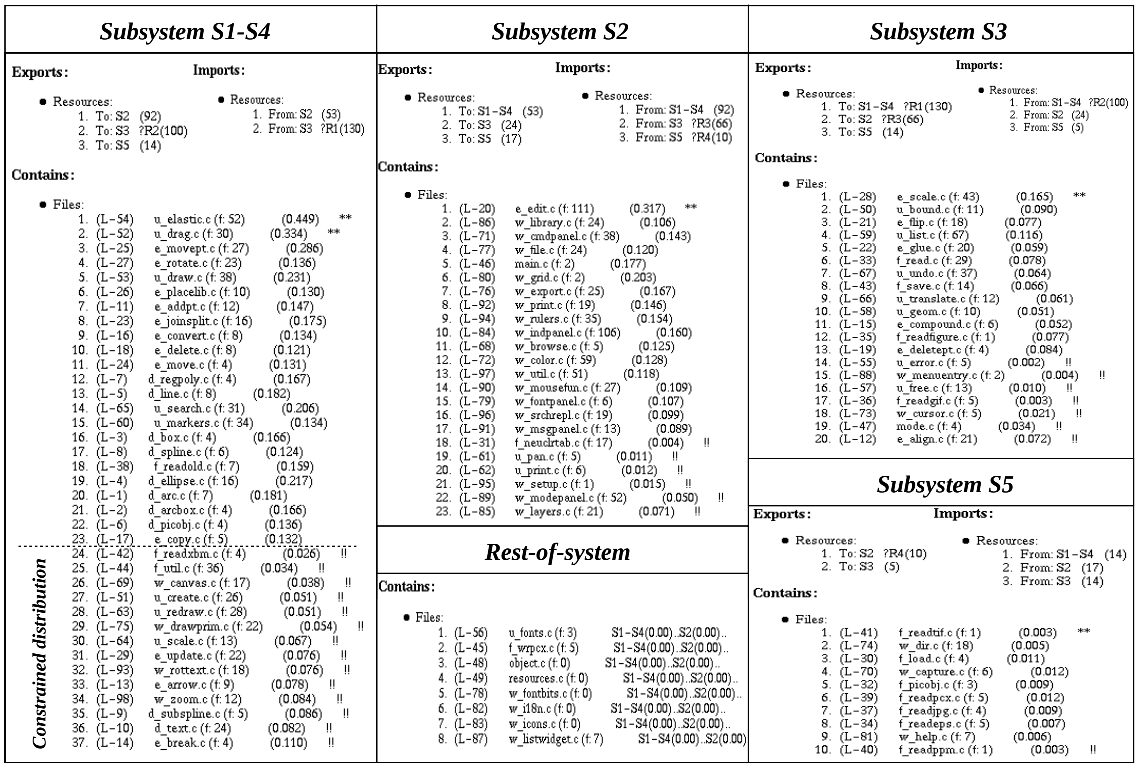
- In order to capture the "file manipulation" operations of the Xfig system into a separate subsystem S5, the main-seed *f_readtif* is selected. It turns out that the windowing and file-manipulation files are recovered in both S2 and S5 which is not a good separation between these two subsystems. Therefore, the interaction between these two subsystem is restricted to a minimum so that while the related files can be grouped in the same subsystem, the interactions are limited.

In general, the assigned constrained links among the subsystems must be limited to important links and a complex pattern with cyclic links be avoided. The user can also change the order of the subsystems so that the most constrained subsystems be recovered first.

In Figured 8.14(a) the hypertext representation of the recovered subsystems using the Netscape browser is illustrated. Each subsystem consists of three parts "Imports", "Exports", and "Contains", where the import/export parts are shown as groups of entities, i.e., *resources*. For example the second line of the "Exports" part of subsystem S1-S4, i.e., "$To: \ S3 \ \ ?R2(100)$", means that subsystem S1-S4 exports 100 functions to subsystem S3 which are the result of matching with the constrained link "?R2(40..100)" in the AQL pattern. The user can also view the individual imported/exported entities and browse their source-code. In this experimentation, the resources are of type function, however the user can select any combination of function, datatype, and variable as the imported/exported resource types. In the "Contains" part, each hypertext line is allocated to one Xfig file. For example, the information in the first line of subsystem S1-S4 is as follows:

$$1. \quad (L\text{-}54) \quad u\_elastic \ (f:52) \qquad (0.449) \qquad **$$

indicating that the main-seed file "*u_elastic*" (symbol "**" means main-seed) with file-id "L-54" has "52" functions and its average similarity value to the remaining 36 files of the subsystem S1-S4 is "0.449". The files that are marked by the symbol "!!" have been assigned to a subsystem during the "constrained distribution" step discussed in Section 8.7.1. The distributed files do not violate the link constrains of the recovered architecture. The files in the *Rest-of-system* are ranked based on



Subsystem S1-S4

Exports:
- Resources:
  1. To: S2 (92)
  2. To: S3 ?R2(100)
  3. To: S5 (14)

Imports:
- Resources:
  1. From: S2 (53)
  2. From: S3 ?R1(130)

Contains:
- Files:

```
 1. (L-54)  u_elastic.c (f: 52)    (0.449)  **
 2. (L-52)  u_drag.c (f: 30)       (0.334)  **
 3. (L-25)  e_movept.c (f: 27)     (0.286)
 4. (L-27)  e_rotate.c (f: 23)     (0.136)
 5. (L-53)  u_draw.c (f: 38)       (0.231)
 6. (L-26)  e_placelib.c (f: 10)   (0.130)
 7. (L-11)  e_addpt.c (f: 12)      (0.147)
 8. (L-23)  e_joinsplit.c (f: 16)  (0.175)
 9. (L-16)  e_convert.c (f: 8)     (0.134)
10. (L-18)  e_delete.c (f: 8)      (0.121)
11. (L-24)  e_move.c (f: 4)        (0.131)
12. (L-7)   d_regpoly.c (f: 4)     (0.167)
13. (L-5)   d_line.c (f: 8)        (0.182)
14. (L-65)  u_search.c (f: 31)     (0.206)
15. (L-60)  u_markers.c (f: 34)    (0.134)
16. (L-3)   d_box.c (f: 4)         (0.166)
17. (L-8)   d_spline.c (f: 6)      (0.124)
18. (L-38)  f_readold.c (f: 7)     (0.159)
19. (L-4)   d_ellipse.c (f: 16)    (0.217)
20. (L-1)   d_arc.c (f: 7)         (0.181)
21. (L-2)   d_arcbox.c (f: 4)      (0.166)
22. (L-6)   d_picobj.c (f: 4)      (0.136)
23. (L-17)  e_copy.c (f: 5)        (0.132)
- - - - - - - - - - - - - - - - - - - - - - -
24. (L-42)  f_readxbm.c (f: 4)     (0.026)  !!
25. (L-44)  f_util.c (f: 36)       (0.034)  !!
26. (L-69)  w_canvas.c (f: 17)     (0.038)  !!
27. (L-51)  u_create.c (f: 26)     (0.051)  !!
28. (L-63)  u_redraw.c (f: 28)     (0.051)  !!
29. (L-75)  w_drawprim.c (f: 22)   (0.054)  !!
30. (L-64)  u_scale.c (f: 13)      (0.067)  !!
31. (L-29)  e_update.c (f: 22)     (0.076)  !!
32. (L-93)  w_rottext.c (f: 18)    (0.076)  !!
33. (L-13)  e_arrow.c (f: 9)       (0.078)  !!
34. (L-98)  w_zoom.c (f: 12)       (0.084)  !!
35. (L-9)   d_subspline.c (f: 5)   (0.086)  !!
36. (L-10)  d_text.c (f: 24)       (0.082)  !!
37. (L-14)  e_break.c (f: 4)       (0.110)  !!
```

Constrained distribution

Subsystem S2

Exports:
- Resources:
  1. To: S1-S4 (53)
  2. To: S3 (24)
  3. To: S5 (17)

Imports:
- Resources:
  1. From: S1-S4 (92)
  2. From: S3 ?R3(66)
  3. From: S5 ?R4(10)

Contains:
- Files:

```
 1. (L-20)  e_edit.c (f: 111)      (0.317)  **
 2. (L-86)  w_library.c (f: 24)    (0.106)
 3. (L-71)  w_cmdpanel.c (f: 38)   (0.143)
 4. (L-77)  w_file.c (f: 24)       (0.120)
 5. (L-46)  main.c (f: 2)          (0.177)
 6. (L-80)  w_grid.c (f: 2)        (0.203)
 7. (L-76)  w_export.c (f: 25)     (0.167)
 8. (L-92)  w_print.c (f: 19)      (0.146)
 9. (L-94)  w_rulers.c (f: 35)     (0.154)
10. (L-84)  w_indpanel.c (f: 106)  (0.160)
11. (L-68)  w_browse.c (f: 5)      (0.125)
12. (L-72)  w_color.c (f: 59)      (0.128)
13. (L-97)  w_util.c (f: 51)       (0.118)
14. (L-90)  w_mousefun.c (f: 27)   (0.109)
15. (L-79)  w_fontpanel.c (f: 6)   (0.107)
16. (L-96)  w_srchrepl.c (f: 19)   (0.099)
17. (L-91)  w_msgpanel.c (f: 13)   (0.089)
18. (L-31)  f_neuclrtab.c (f: 17)  (0.004)  !!
19. (L-61)  u_pan.c (f: 5)         (0.011)  !!
20. (L-62)  u_print.c (f: 6)       (0.012)  !!
21. (L-95)  w_setup.c (f: 1)       (0.015)  !!
22. (L-89)  w_modepanel.c (f: 52)  (0.050)  !!
23. (L-85)  w_layers.c (f: 21)     (0.071)  !!
```

Rest-of-system

Contains:
- Files:

```
1. (L-56)  u_fonts.c (f: 3)        S1-S4(0.00)..S2(0.00)..
2. (L-45)  f_wrpcx.c (f: 5)        S1-S4(0.00)..S2(0.00)..
3. (L-48)  object.c (f: 0)         S1-S4(0.00)..S2(0.00)..
4. (L-49)  resources.c (f: 0)      S1-S4(0.00)..S2(0.00)..
5. (L-78)  w_fontbits.c (f: 0)     S1-S4(0.00)..S2(0.00)..
6. (L-82)  w_i18n.c (f: 0)         S1-S4(0.00)..S2(0.00)..
7. (L-83)  w_icons.c (f: 0)        S1-S4(0.00)..S2(0.00)..
8. (L-87)  w_listwidget.c (f: 7)   S1-S4(0.00)..S2(0.00)
```

Subsystem S3

Exports:
- Resources:
  1. To: S1-S4 ?R1(130)
  2. To: S2 ?R3(66)
  3. To: S5 (14)

Imports:
- Resources:
  1. From: S1-S4 ?R2(100)
  2. From: S2 (24)
  3. From: S5 (5)

Contains:
- Files:

```
 1. (L-28)  e_scale.c (f: 43)      (0.165)  **
 2. (L-50)  u_bound.c (f: 11)      (0.090)
 3. (L-21)  e_flip.c (f: 18)       (0.077)
 4. (L-59)  u_list.c (f: 67)       (0.116)
 5. (L-22)  e_glue.c (f: 20)       (0.059)
 6. (L-33)  f_read.c (f: 29)       (0.078)
 7. (L-67)  u_undo.c (f: 37)       (0.064)
 8. (L-43)  f_save.c (f: 14)       (0.066)
 9. (L-66)  u_translate.c (f: 12)  (0.061)
10. (L-58)  u_geom.c (f: 10)       (0.051)
11. (L-15)  e_compound.c (f: 6)    (0.052)
12. (L-35)  f_readfigure.c (f: 1)  (0.077)
13. (L-19)  e_deletept.c (f: 4)    (0.084)
14. (L-55)  u_error.c (f: 5)       (0.002)  !!
15. (L-88)  w_menuentry.c (f: 2)   (0.004)  !!
16. (L-57)  u_free.c (f: 13)       (0.010)  !!
17. (L-36)  f_readgif.c (f: 5)     (0.003)  !!
18. (L-73)  w_cursor.c (f: 5)      (0.021)  !!
19. (L-47)  mode.c (f: 4)          (0.034)  !!
20. (L-12)  e_align.c (f: 21)      (0.072)  !!
```

Subsystem S5

Exports:
- Resources:
  1. To: S2 ?R4(10)
  2. To: S3 (5)

Imports:
- Resources:
  1. From: S1-S4 (14)
  2. From: S2 (17)
  3. From: S3 (14)

Contains:
- Files:

```
 1. (L-41)  f_readtif.c (f: 1)     (0.003)  **
 2. (L-74)  w_dir.c (f: 18)        (0.005)
 3. (L-30)  f_load.c (f: 4)        (0.011)
 4. (L-70)  w_capture.c (f: 6)     (0.012)
 5. (L-32)  f_picobj.c (f: 3)      (0.009)
 6. (L-39)  f_readpcx.c (f: 5)     (0.012)
 7. (L-37)  f_readjpg.c (f: 4)     (0.009)
 8. (L-34)  f_readeps.c (f: 5)     (0.007)
 9. (L-81)  w_help.c (f: 7)        (0.006)
10. (L-40)  f_readppm.c (f: 1)     (0.003)  !!
```

(a) Recovered subsystems of the Xfig system

Recovered subsystems	No. of files	Xfig subsystems	No. of files	Precision	Recall
S1-S4	37	editing & utility & drawing	47	81%	63% e- 45% 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	5 zero size files	——	——	——
Xfig subsystems:		1) editing: 19 files 2) utility: 18 files 3) drawing: 10 files	4) file manipulation: 16 files 5) X-windowing: 28 files		

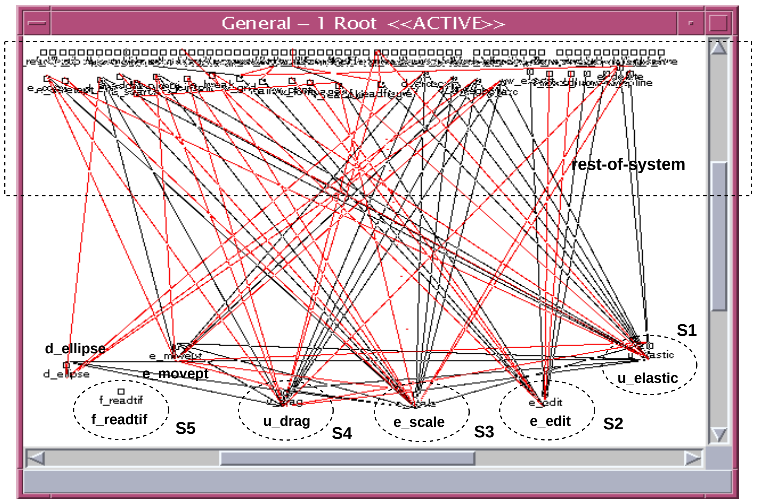
(b) Arcitectural recovery evalution

Figure 8.14: (a) Details of the recovered subsystems using the Netscape browser and the import/export of the system functions as "resources", where both the module sizes and link constrains have been satisfied. (b) Architectural evaluation using "Precision" and "Recall" metrics.

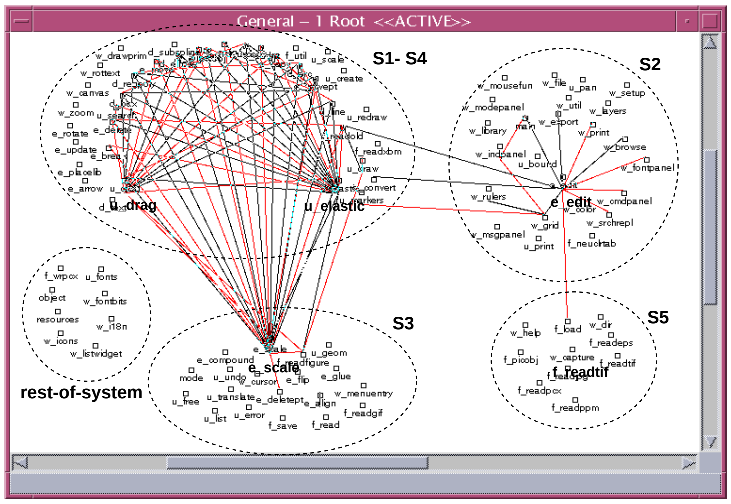
their average similarity values to the recovered subsystems which allow the user to select a subset of the files for the tool to perform constrained distribution. The files that possess very low closeness values to the recovered subsystems will remain in the *Rest-of-system*.

Figure 8.15 illustrates the file association graph feature of the proposed environment for viewing the Xfig recovered architecture. Figure 8.15(a) corresponds to the interaction among the five selected main-seeds and Rest-of-system, where only the strong and medium association links are shown. Figure 8.15(b) illustrates the result of the recovery process, where the highly associated files are grouped into subsystem S1-S4 and the association among the subsystems are limited. Figures 8.15(d) and (e) illustrate the inclusion of the loose and weak association links to Figure 8.15(b), respectively. Figure 8.15(c) 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 8.15(c) 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.

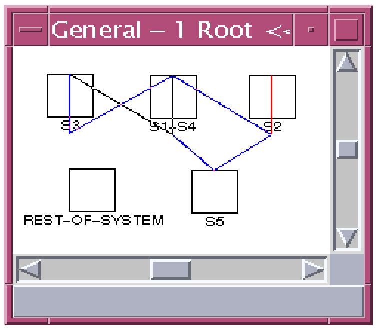
Figure 8.14(b) presents the accuracy of the Xfig recovery process in terms of the Precision and Recall metrics. The subsystem S1-S4 recovers all the drawing files and together with S3 recover almost all the editing and utility files. S2 is allocated to windowing files and S5 recovers file-manipulation files. The obtained Precision and Recall values indicate the accuracy for the proposed pattern matching technique. The AQL query used for the Xfig architectural recovery is presented in



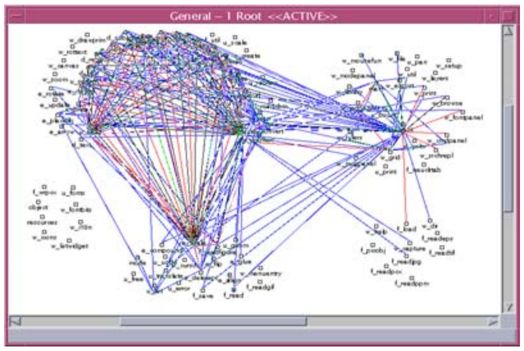
(a) Five Selected main-seeds for five subsystems and the rest-of-system are shown using "strong" and "medium" association links.



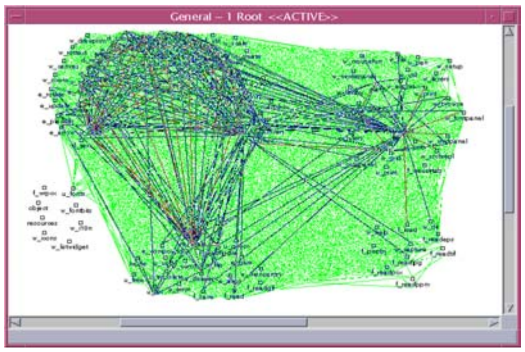
(b) Final recovery of Xfig system: the subsystems S1 and S4 are merged into subsystem S1-S4



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



(d) Adding "loose" association links to part (b)



(e) Adding "weak" association links to part (d)

Figure 8.15: Graph visualization of the recovered subsystems for the Xfig system using the file association graph with different strengths for association links.

Appendix D.

8.7.3 Architecture recovery of Clips

The Clips system provides an environment for implementing rule based expert systems and is supported by an architectural manual [94] which serves as a reference in this experimentation. According to the architectural documentation of the Clips system it consists of 8 "*modules*"[4] as is shown in Figure 8.17(a). The source-code statistics of Clips has been presented in Section 8.1.1. Figure 8.16(a) illustrates the generated architectural pattern for the Clips system with five abstract subsystems and link constrains. The incremental and iterative recovery process yields the recovered architecture in Figure 8.16(b) where the size constrains for the subsystems and links have been satisfied. The hypertext representation of the recovered subsystems using the Netscape browser is shown in Figure 8.17(b), along with the number of matched imported/exported links such as: "*Subsystem S1 Imports From:* $S4 \quad ?R1(20)$".

The pattern for the Clips has been generated based on the tool suggested main-seeds and the documented modules for Clips in Figure 8.17(a). According to the Clips documentation, the recovered subsystem S1 with main-seed file *generate* (corresponding to Clips module 4) should have less interaction with the subsystems S2 and S3 (corresponding to Clips modules 1 and 6, respectively). Therefore, no link constraints are needed between S1 and either S2 or S3. The main-seeds for subsystems S1 and S4 (i.e., *generate* and *expressn*) are contained in the Clips *Parsing*

[4] Please note that the notion of "module" in the Clips documentation is the same as "subsystem" in our terminology.

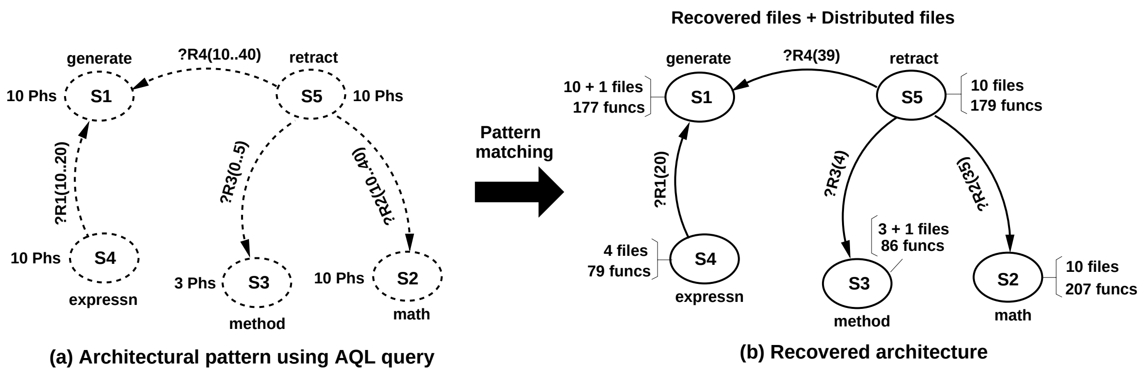
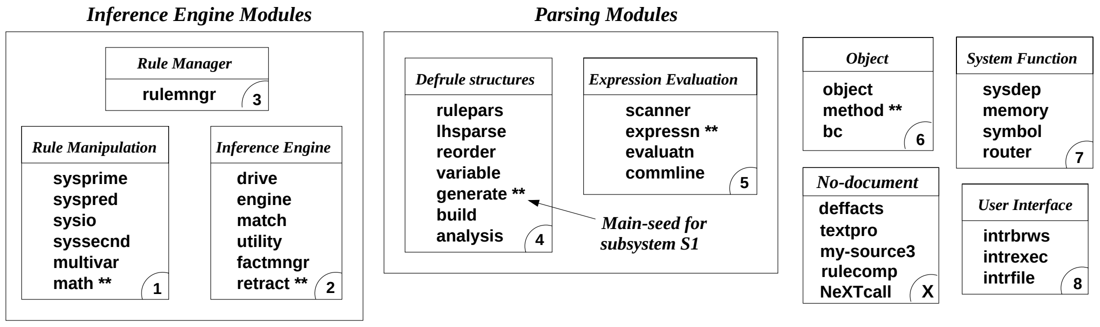


Figure 8.16: (a) The architectural pattern of the Clips system with five abstract subsystems. (b) The recovered architecture where the constraints are satisfied.

modules, hence in order to accumulate related files in both S1 and S4 the link $?R1(10..20)$ is defined whose maximum value 20 imposes more interaction between two subsystems. Also since four of the recovered files in subsystem S1 are from Clips module 2 and subsystem S5 is intended to collect files from module 2, therefore the link $?R4(10..40)$ is defined to attract more files from Clips module 2 to be recovered in subsystem S5. The link $?R2(10..40)$ between S2 and S5 is also defined to collect more related files in the subsystems S2 and S5 which correspond to closely related Clips modules 1 and 2. The subsystem S3 is a rather isolated Clips module with main-seed *method*, however its interaction with subsystem S5 was high. The link $?R3(0..5)$ is defined to restrict this interaction and hence generate a more isolated subsystem.

Figures 8.18(a) and (b) illustrate the graph visualization of the recovered Clips subsystems with the strong and medium association links, where the subsystems S1 and S5 and also S1 and S4 are highly associated and subsystem S3 is isolated. These associations completely conform with the above description for link definition



(a) Clips subsystems according to its architectural documentatoion

Subsystem S1

Exports:
- Resources:
  1. To: S4 (15)
  2. To: S5 (55)
  3. To: S2 (39)
  4. To: S3 (4)

Imports:
- Resources:
  1. From: S4 ?R1(20)
  2. From: S5 ?R4(39)
  3. From: S2 (2)

Contains:
- Files:

					ref
1.	(L−13)	generate.c (f: 20)	(0.595)	**	4
2.	(L−34)	rulepars.c (f: 12)	(0.292)		4
3.	(L−44)	variable.c (f: 13)	(0.243)		4
4.	(L−4)	build.c (f: 6)	(0.374)		4
5.	(L−8)	drive.c (f: 7)	(0.317)		2
6.	(L−12)	factmngr.c (f: 34)	(0.293)		2
7.	(L−9)	engine.c (f: 18)	(0.247)		2
8.	(L−7)	deffacts.c (f: 24)	(0.274)		x
9.	(L−43)	utility.c (f: 35)	(0.277)		2
10.	(L−29)	reorder.c (f: 4)	(0.226)		4
11.	(L−21)	memory.c (f: 4)	(0.144)	!!	7

Subsystem S2

Exports:
- Resources:
  1. To: S1 (2)
  2. To: S4 (1)
  3. To: S5 (10
  4. To: S3 (1)

Imports:
- Resources:
  1. From: S1 (39)
  2. From: S4 (5)
  3. From: S5 ?R2(35)

Contains:
- Files:

					ref
1.	(L−20)	math.c (f: 65)	(0.180)	**	1
2.	(L−24)	multivar.c (f: 9)	(0.200)		1
3.	(L−41)	syssecnd.c (f: 11)	(0.182)		1
4.	(L−38)	sysio.c (f: 21)	(0.158)		1
5.	(L−14)	intrbrws.c (f: 13)	(0.087)		8
6.	(L−42)	textpro.c (f: 24)	(0.078)		x
7.	(L−37)	sysdep.c (f: 12)	(0.049)		7
8.	(L−26)	my_source3.c (f: 4)	(0.024)		x
9.	(L−32)	rulecomp.c (f: 28)	(0.015)		x
10.	(L−39)	syspred.c (f: 20)	(0.082)		1

Subsystem S3

Exports:
- Resources:
  1. To: S5 (16)

Imports:
- Resources:
  1. From: S1 (4)
  2. From: S5 ?R3(4)
  3. From: S2 (1)

Contains:
- Files:

					ref
1.	(L−22)	method.c (f: 25)	(0.440)	**	6
2.	(L−28)	object.c (f: 43)	(0.384)		6
3.	(L−1)	NeXTcall.c (f: 1)	(0.150)		x
4.	(L−3)	bc.c (f: 17)	(0.040)	!!	6

Subsystem S4

Exports:
- Resources:
  1. To: S1 ?R1(20)
  2. To: S5 (6)
  3. To: S2 (5)

Imports:
- Resources:
  1. From: S1 (15)
  2. From: S5 (13)
  3. From: S2 (1)

Contains:
- Files:

					ref
1.	(L−11)	expressn.c (f: 28)	(0.350)	**	5
2.	(L−17)	lhsparse.c (f: 16)	(0.295)		4
3.	(L−35)	scanner.c (f: 20)	(0.190)		5
4.	(L−5)	commline.c (f: 15)	(0.074)		5

Subsystem S5

Exports:
- Resources:
  1. To: S1 ?R4(39)
  2. To: S4 (13)
  3. To: S2 ?R2(35)
  4. To: S3 ?R3(4)

Imports:
- Resources:
  1. From: S1 (55)
  2. From: S4 (6)
  3. From: S2 (10)
  4. From: S3 (16)

Contains:
- Files:

					ref
1.	(L−30)	retract.c (f: 7)	(0.190)	**	2
2.	(L−36)	symbol.c (f: 11)	(0.136)		7
3.	(L−31)	router.c (f: 18)	(0.111)		7
4.	(L−33)	rulemngr.c (f: 15)	(0.093)		3
5.	(L−15)	intrexec.c (f: 38)	(0.091)		8
6.	(L−2)	analysis.c (f: 15)	(0.072)		4
7.	(L−40)	sysprime.c (f: 21)	(0.105)		1
8.	(L−19)	match.c (f: 4)	(0.120)		2
9.	(L−10)	evaluatn.c (f: 19)	(0.080)		5
10.	(L−16)	intrfile.c (f: 31)	(0.073)		8

Rest of System

Contains:
- Files:

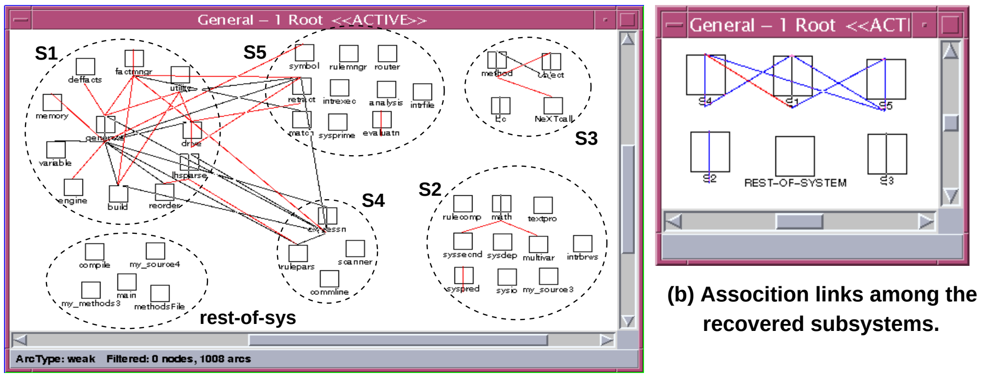
1.	(L−18)	main.c (f: 2)	S3(0.02)..S5(0.00)..
2.	(L−6)	compile.c (f: 3)	S1(0.00)..S4(0.00)..
3.	(L−23)	methodsFile.c (f: 1)	S1(0.00)..S4(0.00)..
4.	(L−25)	my_methods3.c (f: 1)	S1(0.00)..S4(0.00)..
5.	(L−27)	my_source4.c (f: 1)	S1(0.00)..S4(0.00)..

(b) Recovered subsystems of Clips and the reference numbers to the Clips documented subsystems in part (a)

Recovered subsystems	No. of files	Clips subsystems (documented)	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 evaluation	4	75%	75%
S5	10	- System function - User interface	7	40%	57%
rest-of-sys	5	———	——	——	——

(c) Architectural recovery evalution

Figure 8.17: Architectural recovery and evaluation of the Clips system.



ArcType: weak Filtered: 0 nodes, 1008 arcs

(a) Recovered subsystems

(b) Assocition links among the recovered subsystems.

Figure 8.18: Graph visualization of the Clips recovered architecture using "strong" and "medium" association links among system files and among subsystems.

between the subsystems.

Figure 8.17(c) presents the accuracy of the Clips architecture recovery process in terms of Precision and Recall metrics, corresponding to the Clips documentation. Such Precision and Recall values indicate the high accuracy of the recovery process on the Clips system.

8.8 Summary

The design of a sound technique and the development of supporting tools are fundamental requirements for any reverse engineering approach. In this Chapter, the important aspects of the proposed approach to software architecture recovery were tested through a series of experimentations. A reverse engineering toolkit Alborz provides an interactive environment for architecture recovery presented in this thesis. The proposed environment consists of an off-line and an on-line analysis phase.

The set of experimentations for the off-line analysis focused mostly on the extraction of the maximal association property among the system entities using the data mining frequent-itemsets. The time and space complexity for generating associated groups of entities is governed by the density of relationships among the system entities. The important aspects of the pattern matching process including the stability, quality, and accuracy were also tested. The effect of the bounded-queue heuristic in restricting the space complexity and the trade-off for obtaining a tractable matching process with a sub-optimal recovery result were evaluated. The toolkit provides main-seed selection techniques and visualization means to assist in comprehending the structure of the system files and the recovered architecture. Finally, the detailed steps for an incremental and iterative pattern generation and architectural recovery of two middle-size industrial systems were presented.

Chapter 9

Conclusion and future work

This thesis contributes to the reverse engineering research area by providing an interactive environment for architectural recovery, a sound incremental graph pattern matching model of the recovery process, and a prototype toolkit to support the proposed environment. This work uses the techniques from approximate graph matching, data mining, clustering, and architecture description language design.

Architecture recovery technique

In the proposed approximate pattern-matching technique the high-level pattern-graph of the system is matched against a graph representation of the system entities and data-/control-dependencies. During the off-line information extraction phase, the software system is parsed and presented as an attributed relational graph whose nodes and edges conform with an abstract domain model that is suitable for architectural recovery. Such a domain model provides programming language independence for the recovery process as it abstracts away the details of the programming languages. The graph representation of the software system is further processed to allow the recovery of highly associated components, i.e., cohesive components. In order to generate such components, data mining techniques are used to define two association-based similarity metrics between two entities and between two groups of entities.

During the on-line analysis phase, the user generates an architectural pattern (i.e., conceptual architecture) of the system using a proprietary language namely Architecture Query Language (AQL). This language stems from the modular design of the Architecture Description Languages (ADL) to represent system components and their interactions. During the incremental recovery process, the abstract pattern in the AQL query is gradually expanded into a pattern-graph which is matched against an input-graph that is a subset of the system graph. The graph matching process is implemented by a search algorithm that uses both the similarity values among the system entities and their in-between relationships to assign a graph edit cost for each node matching. This pattern matching process is general enough to investigate all possible interactions between two system components using graph-edit operations (i.e., edge insertion/deletion).

In modeling the incremental graph matching approach for architecture recovery, a number of intermediate graphs and connector edges were defined. Such intermediate graphs allowed to represent the pattern-graph and input-graph at each iteration step in terms of their constituents (i.e., a number of recovered modules and their import/export links) and consequently formulate them using recursive graph summation equations. This formulation provides a valuable means for modeling and implementing the whole incremental pattern matching process. Finally, in order to address the tractability issues inherent in the approximate graph pattern matching process, two heuristic techniques are used. The first heuristic divides the graph of the software system into a number of graph regions using the data mining association. The second heuristic reduces the optimal search algorithm A^* used in the pattern matching process into a sub-optimal search BQ-A^*, by restricting the size of queue for the search-tree paths. While this restriction does not necessarily excludes the optimal solution, it reduces the exponential time and space complexity of the search operation.

Also the architecture recovery is performed hierarchically into two levels. First the system is decomposed into a number of subsystems of files. Second each subsystem can be decomposed into a number of modules of functions, datatypes, and variables.

Architecture recovery environment

To experiment the characteristics of the proposed approach a reverse engineering toolkit (Alborz) is implemented. The toolkit provides an interactive environment for recovery and evaluating the architecture of a software system as cohesive components. The toolkit has been built in the Software Refinery environment (Refine) and uses the Refine's built-in parsers to parse the software systems, and built-in parser generator to implement the AQL pattern language. The toolkit provides statistics and metrics to assist in the recovery process. Specifically, the toolkit provides means for visualizing the interaction among the software system files and the recovered modules/subsystems using import/export of entities, and association links.

The challenging step for defining an architectural pattern in the proposed approach corresponds to main-seed selection for the abstract components in the AQL query. The engineer may decide to use one or more of the following methods: i) utilize knowledge about the related domain such as a reference architecture with well-defined components; ii) study the existing design documentation, informal information embedded in the source-code, naming conventions, directory structures; and iii) perform system analysis based on association structure of the system files or different methods of clustering. The adopted method(s) should be able to suggest important and rather distinct main-seeds as the cores of the abstract module/subsystems in the AQL query. The toolkit assists the engineer by providing three main-seed selection techniques based on entity-domain analysis, association-graph analysis, and association-graph visualization.

Finally, in order to assess the usefulness of the proposed approach, a comprehensive set of experiments for off-line and on-line phases were conducted. The experiments focus on evaluating the time/space complexity, stability, quality, accuracy, and tractability of the approach. The generality of the approach was also demonstrated by experimenting with systems in different domains.

9.1 Future work

Possible extension to the work presented in this thesis may focus on four areas.

Clustering: the first extension to this thesis deals with the incorporation of clustering techniques to the matching process. Specifically, an incremental optimization clustering technique may provide a relaxed version of the proposed pattern-matching approach, where there are no constraints on the interaction among the recovered components. Moreover, a partitioning clustering technique may be implemented that starts from an initial partition of singleton clusters and rest-of-system, and performs file relocation between the clusters (subsystems), where the average closeness of a file to the original cluster and to the other clusters is the criteria for file relocation, until the clusters are stable. Finally, an architectural evaluation technique may be used to assess the recovered design of a software system based on different views.

Behavior recovery: the second extension to the thesis deals with behavioral recovery using data mining. The scope of the work in this thesis is limited to structural recovery. However, behavior recovery is a more challenging task that is most often ignored in the current approaches. The existing framework can be augmented to allow behavioral recovery using a data mining technique known as *sequential pattern discovery* [13]. This technique is used to extract relationships among sequences of groups of items in a database of baskets. An application of this technique in extracting a sequence pattern of events in the run-time execution of a program can be exemplified as: 15% of the operations of the subject system, first execute function A, then execute function C, and finally execute function F. The discovery of such a sequence pattern is not affected by any in-between function invocations. The sequential pattern discovery can be applied on an event trace data set obtained from the execution of a software system during its normal operation, in order to recover the highly repeated traces of events. This extension adds an important recovery view of the system to the framework.

Architectural styles: the third extension to this thesis deals with the recovery of *pipe and filter* and *client server* styles. Specifically, the syntax and semantics of the AQL can be enhanced to handle these architectural styles. The AQL language has been designed based on the features of the architecture description languages (ADL) with the objective that it can query the same features that are usually described by the ADL languages. An architectural style consists of the description of the components along with a set of constrains on how they can be connected. For example, the following extensions are required to model the above styles. First, the definition of the similarity metric between entities needs to be revised to include the association (or maximal association) based on new relations such as file read/write (for pipe-and-filter style) and message send/receive (for client-server style). The entity domains should be generated accordingly. Second, the main-seed for a component should be selected according to the expected functionality of the component, e.g., acting as a server, a client, or a filter. The semantics of the connectors (i.e., current import/export parts) must be enhanced to allow the specification of more details about the entities that establish a connection between two components, such as: writing/reading to a specific port or file, using a particular data structure, and importing/exporting entities with constrained naming. The pattern matching process will then be a constraint satisfaction search that allocates entities from the domains to components while satisfies the constraints of the connectors.

Information exchange: finally, another possible extension to this thesis is to adopt a standard information exchange format such as GXL [49] to communicate with a repository of graph-based tools. This standard data interchange format that can be populated by an appropriate parser can be used to receive input entities and relations from a tool and return the recovered results for visualization and further evaluation to appropriate tools.

Appendix A

Formal definitions for domain models

A.1 Source-level domain model

Figure A.1 illustrates a simplified class diagram of the domain model for a typical procedural language. The class attributes of this class diagram are illustrated in Tables A.1 and A.2.

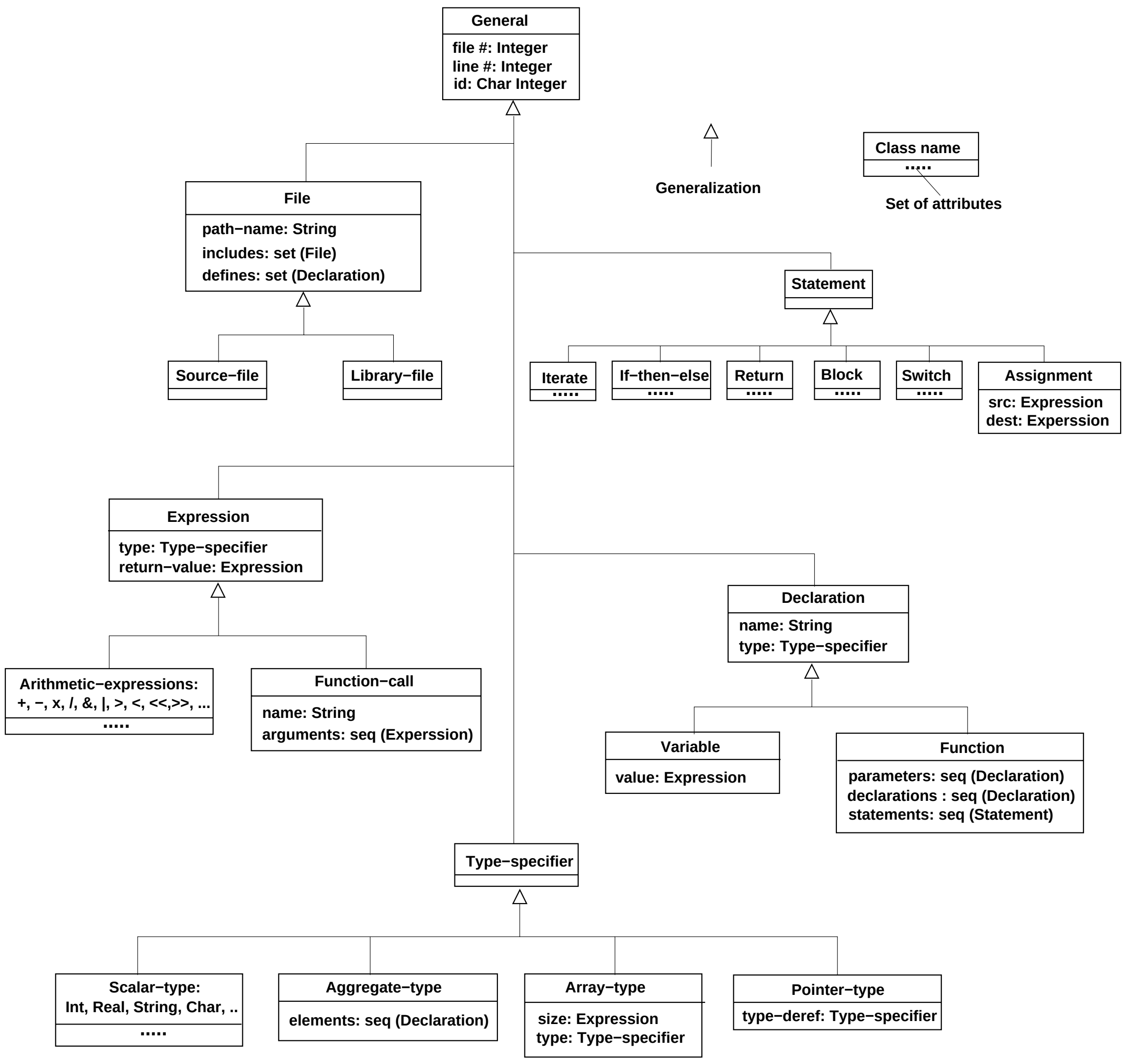


Figure A.1: The class diagram of a simplified *source-level* domain model for a typical procedural programming language such as C [84]. This domain model is used to extract an abstract domain model for architecture-level system analysis. The instantiation of the classes and their association links (not shown here) during the parsing process generates an abstract syntax tree for the software system.

General		
Attribute	*Example*	*Description*
file #	5	File number of the source-code file where entity is defined
line #	79	Line number that an entity appears in a file
id	f4	Unique identifier for each object (entity). The object of an "id" is returned by the function Obj(id).

Source-file / Library-file		
Attribute	*Example*	*Description*
path-name	"/u/../main.c"	Unique path name of the file
includes	{"user.h"}	User-defined library-files whose defined entities are used by the entities in this file
defines		A sequence of functions (for source-file) and a sequence of datatypes and variables (for both source-file and library-file) that are accessible by all entities in this file.

Function		
Attribute	*Example*	*Description*
name	"foo"	Name of the function to be called with
parameters	[x: Integer, ..]	A list of parameters each having a name and a type
declarations	[y: String, ..]	A list of variable and datatype declarations that are local to the function
statements	[x := x + 1; ..]	A list of statements that perform the function's operation
type	String	Type of value that is returned by the function

Declaration		
Attribute	*Example*	*Description*
name	"bar"	Name of the declared function, type, or variable
type	Real	Type-specifier for the declared entity

Table A.1: Description of the class attributes.

Type specifier
Description
Defines the value of a computation or a declaration to be an instance of one of the classes *Scalar-type, Aggregate-type, Array-type*, or *Pointer-type*

Aggregate type		
Attribute	*Example*	*Description*
elements	[field1: String, ..]	List of type declarations for the different fields of the aggregate type

Array type		
Attribute	*Example*	*Description*
size	15	Cardinality of the array entries
type	Integer	Type-specifier for every array entry

Variable		
Attribute	*Example*	*Description*
name	"count"	Name of the defined variable
type	Integer	Type of the variable's value
value	6	Value of the variable

Table A.2: Description of the class attributes.

A.2 Formal definition of relations

In this Section, the relations in the abstract domain model are defined that are categorized into function-level and file-level relations. The Z notation [115, 65] is used to formally define each relation.

In the Z notation, a "set" can be defined as $\{D \mid P \bullet E\}$ denoting a set of values consisting of all values of the term E for the declared variables in D that satisfy the constraint P. The predicate P and term E contain the free variables defined in D. For example, $\{x : \mathcal{N} \mid x \leq 5 \bullet x^2\}$ denotes the set $\{1, 4, 9, 16, 25\}$. The term E and its preceding "heavy dot" can be omitted which results $\{x : \mathcal{N} \mid x \leq 5\} = \{1, 2, 3, 4, 5\}$.

The *existential quantifier* "$\exists$" is used to define a new variable. The general form of the existential quantifier is $\exists D \mid P \bullet Q$ where D represents declarations, P represents a predicate acting as the constraint and Q represents the predicate being quantified. The constraint bar "$|$" and the constraining predicate P can be omitted, which results: $\exists D \bullet Q$.

An existential quantifier $\exists D \mid P \bullet Q$ can be recast as $\exists D \bullet (P \wedge Q)$, and if the predicate Q contains another existential quantifier then the result is $\exists D1 \bullet (P1 \wedge \exists D2 \bullet (P2 \wedge Q2))$

The *universal quantifier* "$\forall$" is used to define all variables that have certain properties. The general form of the universal quantifier is $\forall D \mid P \bullet Q$. The constraint bar "$|$" and the constraining predicate P can be omitted, which results: $\forall D \bullet Q$. A binary relation can be defined as $\{x : T1;\ y : T2 \mid P\}$ which yields a set of binary tuples $\{(x_1, y_1), (x_2, y_2), ...\}$ where each pair (x_i, y_i) satisfies the predicate P.

In the following definitions, the predicate $inherits(a, b)$ indicates that the object a inherits all the attributes of the object b (since class A is a subclass of class B). In Figure A.1, the classes: *Iterate, If-then-else, Return, Block, Switch*, and *Assignment* are subclasses of class *Statement*, each containing one or more attributes whose values are of type Expression (directly or indirectly). These attributes and their "*attribute-values*" are not shown in Figure A.1, however, it is assumed that we can access to these attribute-values.

Function level relations

Relation $\textit{use-F} \subset \textit{Function-abs} \times \textit{Function-abs}$ is defined as:

$$
\begin{aligned}
&\textit{use-F} \;\triangleq\; \{F : \textit{Function-abs};\; F' : \textit{Function-abs} \;\mid \\
&\quad \exists f : Function \bullet (F.\textit{implement-id} = f.id \;\wedge \\
&\quad \exists f' : Function \bullet (F'.\textit{implement-id} = f'.id \;\wedge \\
&\quad \exists s : Statement \bullet (s \in \textit{seq-to-set}(f.statements) \;\wedge \\
&\quad \exists s' : AnyType^{1} \bullet \\
&\qquad (AnyType \in \{Iterate, \textit{If-then-else}, Return, Block, Switch, Assignment\} \;\wedge \\
&\qquad\; inherits(s', s) \;\wedge \\
&\quad \exists x : Expression \bullet (x \in s'.\textit{attribute-values} \;\wedge \\
&\quad \exists fc : \textit{Function-call} \bullet (inherits(fc, x) \;\wedge\; fc.name = f'.name))))))\};
\end{aligned}
$$

$$\forall (F, F') \in \textit{use-F} \implies F' \in F.useFuncs$$

[1] The type *AnyType* can be of any types *Iterate*, *If-then-else*, *Return*, *Block*, *Switch*, *or Assignment*.

Relation $\textit{use-T} \subset \textit{Function-abs} \times \textit{Type-abs}$ is defined as:

$$
\begin{array}{l}
\textit{use-T} \;\triangleq\; \{F : \textit{Function-abs};\; T : \textit{Type-abs} \;\mid \\
\quad \exists f : Function \bullet (F.\textit{implement-id} = f.id \;\wedge \\
\quad \exists t : X;\; \exists t' : \textit{Type-specifier} \bullet (X \in \{\textit{Aggregate-type}, \textit{Array-type}\} \;\wedge \\
\qquad inherits(t, t') \;\wedge\; T.\textit{implement-id} = t.id \;\wedge \\
\quad \exists s : Statement \bullet (s \in \textit{seq-to-set}(f.statements) \;\wedge \\
\quad \exists s' : AnyType \bullet \\
\qquad (AnyType \in \{Iterate, \textit{If-then-else}, Return, Block, Switch, Assignment\} \wedge \\
\qquad inherits(s', s) \;\wedge \\
\quad \exists x : Expression;\;\; \exists v : Variable \bullet x \in s'.\textit{attribute-values} \;\wedge \\
\qquad\qquad (v.value = x \;\vee\; x = v.value) \;\wedge\; v.type = t))))\}; \\
\forall (F, T) \in \textit{use-T} \implies T \in F.useTypes
\end{array}
$$

Relation $\textit{use-V} \subset \textit{Function-abs} \times \textit{Variable-abs}$ is defined as:

$$
\begin{array}{l}
\textit{use-V} \;\triangleq\; \{F : \textit{Function-abs};\; V : \textit{Variable-abs} \;\mid \\
\quad \exists f : Function \bullet (F.\textit{implement-id} = f.id \;\wedge \\
\quad \exists v : Variable;\;\; \exists d : Declaration;\;\; \exists l : File \bullet \\
\qquad (inherits(v, d) \;\wedge\; v \in l.defines \;\wedge\; V.\textit{implement-id} = v.id \;\wedge \\
\quad \exists s : Statement \bullet (s \in \textit{seq-to-set}(s.statements) \;\wedge \\
\quad \exists s' : AnyType \bullet \\
\qquad (AnyType \in \{Iterate, \textit{If-then-else}, Return, Block, Switch, Assignment\} \wedge \\
\qquad inherits(s', s) \;\wedge \\
\quad \exists x : Expression \bullet x \in s'.\textit{attribute-values} \;\wedge \\
\qquad (v.value = x \;\vee\; x = v.value)))))\};
\end{array}
$$

$$\forall (F, V) \in \textit{use-V} \implies V \in F.useVars$$

File level relations

Relation $\textit{cont-R} \subset \textit{File-abs} \times \textit{Entity-abs}$ (contain-resource) is defined as:

$$\begin{aligned}
&\textit{cont-R} \triangleq \\
&\quad \{L : \textit{File-abs};\ F : \textit{Function-abs} \mid \\
&\quad\quad \exists ls^2 : \textit{Source-file} \bullet (L.\textit{implement-id} = ls.id \wedge \\
&\quad\quad \exists f : Function;\ \ \exists d : Declaration \bullet inherits(f, d) \wedge \\
&\quad\quad f \in ls.defines \wedge F.\textit{implement-id} = f.id)\} \\
&\quad \cup \\
&\quad \{L : \textit{File-abs};\ T : \textit{Type-abs} \mid \\
&\quad\quad \exists ls : \textit{Source-file} \bullet (L.\textit{implement-id} = ls.id \wedge \\
&\quad\quad \exists t : AnyType;\ \exists t' : \textit{Type-specifier};\ \exists d : Declaration \bullet \\
&\quad\quad\quad (AnyType \in \{\textit{Aggregate-type}, \textit{Array-type}\} \wedge inherits(t, t') \wedge \\
&\quad\quad\quad d \in ls.defines \wedge d.type = t \wedge T.\textit{implement-id} = t.id)\} \\
&\quad \cup \\
&\quad \{L : \textit{File-abs};\ V : \textit{Variable-abs} \mid \\
&\quad\quad \exists ls : \textit{Source-file} \bullet (L.\textit{implement-id} = ls.id \wedge \\
&\quad\quad \exists v : Variable;\ \ \exists d : Declaration \bullet d \in ls.defines \wedge \\
&\quad\quad inherits(v, d) \wedge V.\textit{implement-id} = v.id)\};
\end{aligned}$$

$$\forall (L, R) \in \textit{cont-R} \implies R \in L.contains$$

[2] Every source-file ls inherits the attributes of its superclass file l as: $\exists l : File;\ \exists ls : \textit{Source-file} \bullet inherits(ls, l)$.

Relation $\mathit{use\text{-}R} \subset \mathit{File\text{-}abs} \times \mathit{Entity\text{-}abs}$ (use-resource) is defined as:

$$\begin{aligned}
&\mathit{use\text{-}R} \triangleq \\
&\quad \{L : \mathit{File\text{-}abs};\ \ F : \mathit{Function\text{-}abs} \mid \\
&\qquad \exists F' : \mathit{Function\text{-}abs} \bullet (L, F') \in \mathit{cont\text{-}R} \ \wedge\ (F', F) \in \mathit{use\text{-}F}\} \\
&\quad \cup \\
&\quad \{L : \mathit{File\text{-}abs};\ \ T : \mathit{Type\text{-}abs} \mid \\
&\qquad \exists F' : \mathit{Function\text{-}abs} \bullet (L, F') \in \mathit{cont\text{-}R} \ \wedge\ (F', T) \in \mathit{use\text{-}T}\} \\
&\quad \cup \\
&\quad \{L : \mathit{File\text{-}abs};\ \ V : \mathit{Variable\text{-}abs} \mid \\
&\qquad \exists F' : \mathit{Function\text{-}abs} \bullet (L, F') \in \mathit{cont\text{-}R} \ \wedge\ (F', V) \in \mathit{use\text{-}V}\};
\end{aligned}$$

$$\forall (L, R) \in \mathit{use\text{-}R} \implies R \in L.uses$$

Relation $\mathit{imp\text{-}R} \subset \mathit{File\text{-}abs} \times \mathit{Entity\text{-}abs}$ (import-resource) is defined as:

$$\begin{aligned}
&\mathit{imp\text{-}R} \triangleq \{L : \mathit{File\text{-}abs};\ \ R : \mathit{Entity\text{-}abs} \mid \\
&\qquad (L, R) \in \mathit{use\text{-}R} \ \wedge\ (L, R) \notin \mathit{cont\text{-}R}\};
\end{aligned}$$

$$\forall (L, R) \in \mathit{imp\text{-}R} \implies R \in L.imports$$

Relation $\mathit{exp\text{-}R} \subset \mathit{File\text{-}abs} \times \mathit{Entity\text{-}abs}$ (export-resource) is defined as:

$$\begin{aligned}
&\mathit{exp\text{-}R} \triangleq \{L : \mathit{File\text{-}abs};\ \ R : \mathit{Entity\text{-}abs} \mid \\
&\qquad (L, R) \in \mathit{cont\text{-}R} \ \wedge\ \exists L' : \mathit{File\text{-}abs} \bullet (L', R) \in \mathit{use\text{-}R}\};
\end{aligned}$$

$$\forall (L, R) \in \mathit{exp\text{-}R} \implies R \in L.exports$$

Appendix B

Graph definitions

This Appendix presents the formal definition of the graphs introduced in Chapter 5.

B.1 Query graph

The *query-graph* $G^q = (N^q, R^q, A^q, E^q, \mu^q, \epsilon^q)$ or simply $G^q = (N^q, R^q)$ is a *multigraph*[1] [53, 41, 102] with *composite* nodes (query-nodes) and *composite* edges (query-edges). Formally:

- N^q : $\{qn_1, qn_2, ..., qn_l\}$ is the set of attributed nodes in the query-graph.
- R^q : $\{qr_1, qr_2, ..., qr_w\}$ is the set of attributed edges in the query-graph such that $R^q = \{qr_k \mid qr_k = (qn_u, qn_i, t) \wedge qn_u, qn_i \in N^q \wedge t \in \{\textit{use-F},\ \textit{use-T},\ \textit{use-V}\}\}$, where t is a type attribute for graph edges.

[1]In a *multigraph*, more than one edge in each direction between a pair of nodes are allowed.

- A^q: alphabet for node attributes and node attribute values, e.g., node-types and cardinalities.
- E^q: alphabet for edge attributes and edge attribute values, e.g., edge types and cardinalities.
- $\mu^q : N^q \rightarrow (A^q \times A^q)^v$ a function for returning "node attribute, node attribute value" pairs, where v is a constant.
- $\epsilon^q : R^q \rightarrow (E^q \times E^q)^w$ a function for returning "edge attribute, edge attribute value" pairs, where w is a constant.

Query nodes: each *query-node* qn_i is an instance of the class *Module* at function-level (or class *Subsystem* at file-level) analysis in the AQL domain model of Figure 4.2 in Section 4.2.1. At function-level a query-node qn_i denotes three groups of nodes with types *Function-abs, Type-abs, Variable-abs.* The major attributes for each node include: i) a name corresponding to the AQL query module; ii) one or more distinguished nodes (called *main-seeds*), where each *main-seed* n_j associates a source-region G_j^{sr} with the query-node qn_i; iii) zero or more known nodes from the source region G_j^{sr} (called *seeds*); iv) a group-id that represents a set of placeholder-nodes with a cardinality within the range (*min, max*) to be matched with source-graph nodes.

The node labeling function $\mu^q(qn_i)$ returns the "attribute, attribute value" pairs of the query-node qn_i pertaining to all three types in the query-node. An example of the node labeling function is as follows:

$\mu^q(qn_1) = ((\text{name}, \text{“}M1\text{”}),$

$(\text{mainSeed}_F, \text{“}/../foo\text{”}),\ (\text{seed}_F, \text{“}/../bar\text{”}), (\text{groupId}_F, \$\text{CF}),\ (\text{min}_F, 5),\ (\text{max}_F, 9),$
$(\text{mainSeed}_T, \text{“}/../tp1\text{”}),\ (\text{groupId}_T, \$\text{CT}),\ (\text{min}_T, 3),\ (\text{max}_T, 6),$
$(\text{mainSeed}_V, \text{“}/../vr1\text{”}),\ (\text{seed}_V, \text{“}/../vr2\text{”}), (\text{groupId}_V, \$\text{CV}),\ (\text{min}_V, 2),\ (\text{max}_V, 4)$

The above attribute values are interpreted as: the query-node number 1 with name $M1$ identifies three groups of placeholder-nodes. The group of placeholder-nodes with type *Function-abs* are identified with the group-id $CF whose size is restricted between minimum 5 and maximum 9, and two placeholder-nodes have been initially matched with the main-seed node “*foo*” and seed node “*bar*”. Similar explanation can be given for the other two groups of placeholder-nodes $CT and $CV.

Query edges: each *query-edge* qr_k is an instance of the class *Conn-placeholders* in the AQL domain model of Figure 4.2 in Section 4.2.1. A query-edge qr_k denotes a group of edges with a particular edge type $t \in \{$*use-F, use-T, use-V*$\}$ between two query-nodes qn_u and qn_i. The major attributes of a query-edge include: i) source and sink query-nodes (i.e., modules) for the group of edges; ii) a group-id for the group of edges and a cardinality range (*min, max*).

The edge labeling function $\epsilon^q(qr_k)$ returns the “attribute, attribute value” pairs of the query-edge qr_k. An example of the edge labeling function is as follow:

$qr_2 = (qn_4, qn_1, \textit{use-F})$
$\epsilon^q(qr_2) = ((from, qn_4),\ (to, qn_1),\ (type, \textit{use-F}),\ (groupId, ?\text{F1}),\ (min, 3),\ (max, 7)).$

The above attribute values are interpreted as: the query-edge number 2 identifies a group of graph edges of type *use-F* and the group-id ?F1 that emanate

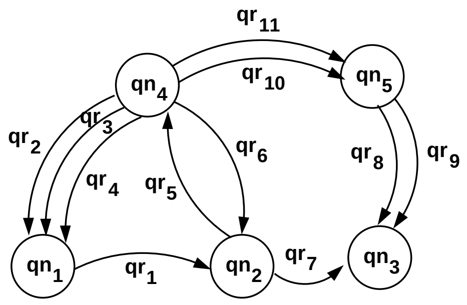


Figure B.1: A query-graph with five query-nodes and eleven query-edges.

from the source-graph nodes (represented by qn_4) and point to the source-graph nodes (represented by qn_1) whose size is restricted between minimum 3 and maximum 7.

Figure B.1 illustrates a query-graph consisting of five query-nodes and eleven query-edges, where at most three query-edges (i.e., between qn_4 and qn_1) can exist between a pair of query-nodes in each direction.

B.2 Pattern-region

The *pattern-region* graph at matching phase i is an ARG defined as:
$G_i^{pr} = (N_i^{pr}, R_i^{pr}, A^{pr}, E^{pr}, \mu^{pr}, \epsilon^{pr})$, or simply $G_i^{pr} = (N_i^{pr}, R_i^{pr})$, corresponding to a composite-node qn_i of the query-graph G^q. At phase i, the pattern-region G_i^{pr} is partially matched against the selected source-region $G_{g(i)}^{sr}$. Graph G_i^{pr} is defined as:

- N_i^{pr} : $\{n_{i,1}, n_{i,2}, ..., n_{i,z}\}$ set of attributed *placeholder-nodes* that correspond to a query-node qn_i, where z is the total size of the placeholders in qn_i. Each placeholder-node will be matched against a node from the selected source-region $G^{sr}_{g(i)}$.

- R^{pr}_i : $\{r_1, r_2, ..., r_m\}$ set of attributed edges,
- $A^{pr} \subseteq A^s \cup \{nil\}$: source-graph node attributes and node attribute values for the matched placeholder-nodes and *nil* label for the yet not-matched placeholder-nodes.
- $E^{pr} \subseteq E^s$: source-graph edge attributes and edge attribute values for the matched edges and type for yet not-matched edges.
- $\mu^{pr} \subseteq \mu^s$ and $\epsilon^{pr} \subseteq \epsilon^s$: functions for returning "node attribute, node attribute value" pairs and "edge attribute, edge attribute value" pairs, respectively.

The steps for generating a pattern-region G^{pr}_i from a query-node qn_i are as follows:

Step A (generic nodes): for each type $t \in$ {*Function-abs, Type-abs, Variable-abs*} and with size-range $(min_t,\ max_t)$ in qn_i, generate as many as max_t (e.g., max_F for type *Function-abs*) placeholder-nodes $n_{i,k}$'s with node-type t and *nil* labels. For each placeholder-node $n_{i,k}$, i is the matching phase number and k is an ordering number which first enumerates all placeholder-nodes of type *Function-abs* (F), then enumerates all placeholder-nodes of type *Type-abs* (T), and finally, enumerates all placeholder-nodes of type *Variable-abs* (V).

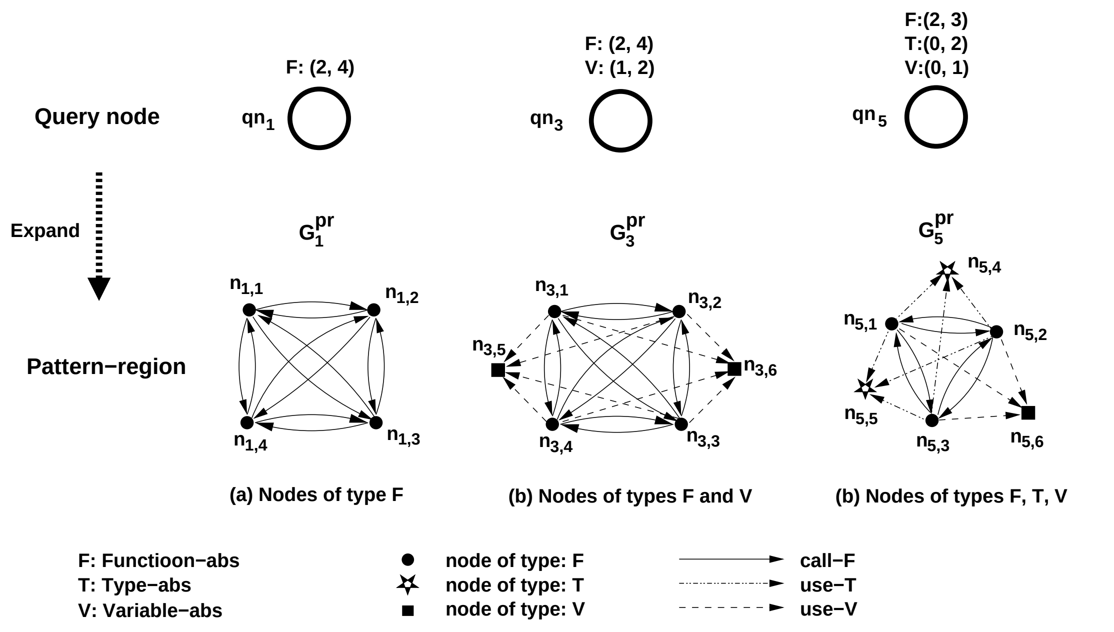


Figure B.2: (a) Three query-nodes with different node-types and their corresponding pattern-regions.

Step B (generic edges): connect every placeholder-node of type F to every other placeholder-node of type F, T, and V, using the edge types *use-F*, *use-T*, and *use-V*, respectively. This yields a fully connected graph with respect to the allowed edges between nodes.

Figure B.2 illustrates the expansion of three query-nodes with different types into three pattern-regions. For example, qn_1 is expanded into four nodes, i.e., its maximum number of nodes, which are fully connected to generate G_1^{pr}.

B.3 Graph connectors and graph summation

Edge-bundle

An *edge-bundle* is a group of edges corresponding to the partial expansion of a query-edge qr_k at phase i of the matching process. Two cases exist as follows:

- *Imported edge-bundle* bi_x: each single edge in the composite query-edge $qr_k = (qn_u, qn_i, t)$ generates an imported edge-bundle bi_x, where, the edges in bi_x connect every node with type t_1 in the matched-region G_u^{mr} to one node with type t_2 (i.e., *sink-node*) in the pattern-region G_i^{pr}, such that the triple (t_1, t, t_2) is a valid triple (*node-type, edge-type, node-type*) in the source-graph G^s. For example, if in a query-edge qr_z the maximum number of edges is 5, then $x \in [1..5]$ and five imported edge-bundles $bi_1, bi_2,\ ...,\ bi_5$ are generated.

- *Exported edge-bundle* be_y: each single edge in the composite query-edge $qr_k = (qn_i, qn_u, t)$ generates an exported edge-bundle be_y, where, the edges in be_y connect one node with type t_1 (i.e., *source-node*) in the pattern-region G_i^{pr} to every node with type t_2 in the matched-region G_u^{mr}, such that the triple (t_1, t, t_2) is a valid triple (*node-type, edge-type, node-type*) in the source-graph G^s.

Connector edges

Formally, the group of connector edges $\mathcal{R}^{G_1 \leftrightarrow G_2}$ is defined between two graphs, as:

Let $G1 = (N1, R1);\ \ G2 = (N2, R2)$ then

$\mathcal{R}^{G_1 \leftrightarrow G_2} = \{x; y \mid (x \in N_1 \ \wedge \ y \in N_2) \ \ \vee \ \ (x \in N_2 \ \wedge \ y \in N_1)\}$

Graph summation

Formally, the binary operator sum "+" is defined between two graphs, as:

Let $G = (N, R)$; $G1 = (N1, R1)$; $G2 = (N2, R2)$ then

$G = G1 + G2$ iff

$N = N1 \cup N2 \wedge R = R1 \cup R2$

Formally, the binary operator o-plus "$\oplus$" is defined between a graph and a set of edges, as:

Let $G = (N, R)$; $G1 = (N1, R1)$; $G^s = (N^s, R^s)$; $R' \subset R^s$ then

$G = G1 \oplus R'$ iff

$N = N1 \cup \{n \mid \exists x \in N^s \bullet (n, x) \in R' \vee (x, n) \in R'\}$;

$R = R1 \cup R'$

B.4 Expanding query-edge into edge-bundles

The steps for expanding a query-edge qr_k into a number of edge-bundles are as follows: consider the query-node qn_i and for every imported query-edge $qr_k = (qn_u, qn_i, t)$ or exported query-edge $qr_k = (qn_i, qn_u, t)$ where $u < i$ do:

1. Using the edge-type t in qr_k ($t \in \{$*use-F, use-T, use-V*$\}$) identify the pair of allowed node-types (t_1, t_2) for the source and sink nodes in pattern-region G_i^{pr} and link module G_u^{mr}, where, $t_1, t_2 \in \{$*Function-abs, Type-abs, Variable-abs*$\}$.

2. Based on the discussion on Section B.3 generate p (max size qr_k) edge-bundles with edge type t between p nodes of G_i^{pr} and all nodes of G_u^{mr} according to the

direction of qr_k. Initially, the first p nodes in the pattern-region are selected as the source-nodes or sink-nodes of the edge-bundles.

Appendix C

Algorithms

Algorithm control-iterative-recovery (G^q, $entType$, S) =

Description:
This algorithm generates an iterative matching-process consisting of i phases ($i \in [1..|N^q|]$). The control mechanism for advancing to the next phase or backtracking to the previous phase is illustrated in Figure 7.1(d), where the control mechanism maintains a list of multi-phase search-trees $LQ\mathcal{G}^p$ and invokes the $BQ\text{-}A^*$ search algorithm at different phases to generate the search-trees.

Input:
G^q: query-graph (N^q, R^q) used for creating and initializing multi-phase search-trees at different matching phases $i \in [1..|N^q|]$.
entType: the type of entities to be recovered in the components.
S: *system representation* as the tuple $(G^s,\ D(N^s))$.

Output:
result of the recovery process as either matched-graph $\mathcal{G}^m_i$ or Nil.

Local variables:
$Q\mathcal{G}^p_i$, $newQ\mathcal{G}^p_i$: multi-phase search-tree for current phase i, implemented as a queue of incomplete multi-phase tree-paths $\mathcal{G}^p_i$.
$LQ\mathcal{G}^p$: list of all multi-phase search-trees from phase 1 to i.
result: result of the search algorithm as matched-graph $\mathcal{G}^m_i$ or Nil.

Continued in the next page ....

```
$LQ\mathcal{G}^p := [\,]$    $i := 1$    $result := Nil$    $stop := false$
while  $stop = false$  do
   if  $|LQ\mathcal{G}^p| < i$  then
      $(Q\mathcal{G}_i^p, \mathcal{G}_{g(i)}^{sr'}) := create\text{-}and\text{-}initialize\text{-}tree(G^q,\ i, result,\ S)$
      $LQ\mathcal{G}^p := append(LQ\mathcal{G}^p, Q\mathcal{G}_i^p)$

   else-if  $|LQ\mathcal{G}^p| = i$  then
      $Q\mathcal{G}_i^p := LQ\mathcal{G}^p[i]$
      $(newQ\mathcal{G}_i^p, result) := BQ\text{-}A^*(Q\mathcal{G}_i^p,\ \mathcal{G}_{g(i)}^{sr'},\ i,\ S)$
       $LQ\mathcal{G}^p[i] := newQ\mathcal{G}_i^p$

       if  $result \neq Nil \ \wedge \ i = |N^q|$  then
          $stop := ture$                              %(SOLUTION)
       else-if  $result \neq Nil$  then
          $i := i + 1$                                %(GO NEXT PHASE)

       else-if  $result = Nil$  then
          if  $i > 1$  then
             $LQ\mathcal{G}^p := delete\text{-}list\text{-}entry(LQ\mathcal{G}^p, i)$
             $i := i - 1$                             %(BACKTRACK)
          else-if  $i = 1$  then
             $stop := true$                           %(FAIL)
 return  $result$
```

Figure C.1: Algorithm: control iterative-recovery

Algorithm *BQ-A** $(Q\mathcal{G}_i^p,\ \mathcal{G}_{g(i)}^{sr'},\ i,\ S) =$

Description:
At each matching phase i, the *BQ-A** algorithm receives a multi-phase search-tree that can be just created-and-initialized (i.e., starting a new phase i), or can be a partially expanded multi-phase search-tree (i.e., backtracking has occurred). The algorithm expands the search-tree to generate a matched-graph as a result, and returns both the result and the search-tree which is further expanded.

Input:
$Q\mathcal{G}_i^p$, $newQ\mathcal{G}_i^p$: multi-phase search-tree for current phase i,
implemented as a queue of incomplete multi-phase tree-paths $\mathcal{G}_i^p$.
$\mathcal{G}_{g(i)}^{sr'}, i$: search-space for current phase i.
S: system representation as the tuple $(G^s,\ D(N^s))$.

Output:
if search succeeded then return solution $\mathcal{G}_i^m$ and expanded multi-phase search-tree $Q\mathcal{G}_i^p$, otherwise return *Nil* and empty search-tree.

Local variables:
$N_{i_{mch}}^{pr}$: set of already matched nodes in $\mathcal{G}_i^{pr}$.
$cost_{mch}$: cost of matching node n_k with placeholder-node $n_{i,j}$.
$cost_{path}$: accumulated cost of matching nodes/edges in $\mathcal{G}_i^{pr}$.
$cost_{uest}$: under-estimated cost of matching the remaining nodes in $\mathcal{G}_i^{pr}$.
$cost$: total estimated cost of matching to get to an optimal solution.
$\mathcal{G}_i^m$: matched-graph as solution of the search at phase i.

Global variables:
$History$: list of incomplete tree-paths $\mathcal{G}_i^{pr}$'s, used for repeated state checking.

Continued in the next page ....

```
found := false
while  $Q\mathcal{G}_i^p \neq [\,] \wedge \neg found$  do
   $\mathcal{G}_i^p := dequeue(Q\mathcal{G}_i^p)$                % get lowest-cost pattern-graph.
   $(\mathcal{G}_i^{pr}, (\mathcal{R}_i^{mr^* \rightarrow pr_i},\ \mathcal{R}_i^{mr^* \leftarrow pr_i})) := \mathcal{G}_i^p[i]$
   $n_{i,j} := current\text{-}placeholder\text{-}node(\mathcal{G}_i^{pr})$      % get matching info.
   $N_{i_{mch}}^{pr} := matched\text{-}nodes(\mathcal{G}_i^{pr})$
   $cost_{path} := cost\text{-}of\text{-}path(\mathcal{G}_i^{pr})$

   if  $n_{i,j} = last\text{-}placeholder\text{-}node(N_i^{pr})$  then
      $\mathcal{G}_i^m := \mathcal{G}_i^p$
      $found := true$

    else
      for  $n_k \in \mathcal{G}_{g(i)}^{sr'}$  do
         if  not-repeated($History$, ($N_{i_{mch}}^{pr}$ with $n_k$))  then
            $(new\mathcal{G}_i^p, cost_{mch}) := evaluate\text{-}node\text{-}matching\text{-}cost(\mathcal{G}_i^p,\ n_k,\ i,\ S)$
            $cost_{path} := cost_{path} + cost_{mch}$
            $cost := cost_{path} + cost_{uest}$
            if  $cost < maxCost$  then
               $\mathcal{G}_i^p$ [i] $:= update\text{-}pathCost\ (new\mathcal{G}_i^p[i],\ cost_{path})$
               $History := add\text{-}to\text{-}history(History, new\mathcal{G}_i^p)$
               $Q\mathcal{G}_i^p := enqueue\text{-}path(Q\mathcal{G}_i^p,\ new\mathcal{G}_i^p)$
               $Q\mathcal{G}_i^p := sort(Q\mathcal{G}_i^p,\ cost)$      % put new $\mathcal{G}_i^p$ in proper slot
            else
               "costly $new\mathcal{G}_i^p$ is discarded"
         else
            "repeated $\mathcal{G}_i^p$ is discarded"

 if  $found$  then
    return  $(Q\mathcal{G}_i^p,\ \mathcal{G}_i^m)$
 else
    return  $([\,],\ Nil)$
```

Figure C.2: Algorithm: *BQ-A**

Algorithm evaluate-node-matching-cost ($\mathcal{G}_i^p,\ n_x,\ i,\ S$) =

Description:
This algorithm combines two algorithms *evaluate-node-matching-cost* and *inside-edge-deletion-cost* discussed in Section 7.2. The algorithm matches the current node n_x from the search-space $\mathcal{G}_{g(i)}^{sr'}$ with a placeholder-node $n_{i,j}$ from the pattern-region $\mathcal{G}_i^{pr}$, and evaluates the overall graph-edit cost for such matching. For each node-matching pair $(n_{i,j}, n_x)$ in $\mathcal{G}_i^{pr}$, the edge-matching is performed in two steps: i) inside-edge matching for $\mathcal{G}_i^{pr}$ using the edge-deletion cost c_{in}^{ed} discussed in Section 6.4.2; and ii) connector-edge matching separately for imported and exported connector-edges using the costs discussed in Section 6.4.2.

Input:
$\mathcal{G}_i^p$, i: multi-phase tree-path (i.e., pattern-graph) at phase i.
n_x: node to be matched from search-space $\mathcal{G}_{g(i)}^{sr'}$.
S: system representation as the tuple $(G^s,\ D(N^s))$.

Output: pattern-graph $\mathcal{G}_i^p$ (such that the pair of nodes $(n_{i,j}, n_x)$ in $\mathcal{G}_i^{pr}$ have been matched) along with the cost of matching $cost_{mch}$.

Local variables:
$N_{i_{mch}}^{pr}$: set of already matched nodes in $\mathcal{G}_i^{pr}$.
k: number of already matched nodes in $\mathcal{G}_i^{pr}$.
e_F, e_R: forward and reverse edges between two nodes n_x and n_y in G^s.
M: max similarity value between two nodes in the search-space $\mathcal{G}_{g(i)}^{sr'}$
c_{in}^{ed}: cost of inside-edge deletion.
c_{imp}, c_{exp}: cost of matching for imported and exported connector-edges.
$cost_{mch}$: cost of matching node n_x with placeholder-node $n_{i,j}$.

Continued in the next page ....

```
% retrieve $\mathcal{G}_i^{pr}$ and its attributes.
$(\mathcal{G}_i^{pr},\ (\mathcal{R}_i^{mr^*\rightarrow pr},\ \mathcal{R}_i^{mr^*\leftarrow pr})) := \mathcal{G}_i^p[i]$
$n_{i,j}$ := $get\text{-}new\text{-}unmatched\text{-}node(\mathcal{G}_i^{pr})$
$N_{i_{mch}}^{pr}$ := $get\text{-}matched\text{-}nodes(\mathcal{G}_i^{pr})$
$k := |N_{i_{mch}}^{pr}|$
$c_{in}^{ed} := 0.0$    $c_{imp} := 0.0$    $c_{exp} := 0.0$
% get cost $c_{in}^{ed}$ based on similarity values and existing edges.
for $n_y \in N_{i_{mch}}^{pr}$ do

    $e_F := (n_x, n_y)$    $e_R := (n_y, n_x)$    $s := entAssoc(n_x, n_y)$

    if $e_F \in R^s \wedge e_R \in R^s$ then    % zero edge deletion cost
        $c_{in}^{ed} := c_{in}^{ed} + \frac{M-s}{k}$

    else-if $e_F \in R^s \vee e_R \in R^s$ then    % one edge deletion cost
        $c_{in}^{ed} := c_{in}^{ed} + \frac{M-0.75s}{k}$

    else-if $e_F \notin R^s \wedge e_R \notin R^s$ then  % two edge deletion cost
        $c_{in}^{ed} := c_{in}^{ed} + \frac{M-0.5s}{k}$

 % append the new matching pair $(n_{i,j}, n_x)$ to the tree-path $\mathcal{G}_i^{pr}$.
 $\mathcal{G}_i^{pr}$ := $append\text{-}to\text{-}matching\text{-}pairs\ (\mathcal{G}_i^{pr}, (n_{i,j}, n_x))$
 $\mathcal{G}_i^p[i] := \mathcal{G}_i^{pr}$

 % get costs $c_{imp}$ and $c_{exp}$ for phase two and higher.
 if $i > 1$ then.
    $(\mathcal{G}_i^p, c_{imp}) := import\text{-}edge\text{-}matching\text{-}cost(\mathcal{G}_i^p,\ n_x,\ i,\ S)$
    $(\mathcal{G}_i^p, c_{exp}) := export\text{-}edge\text{-}matching\text{-}cost(\mathcal{G}_i^p,\ n_x,\ i,\ S)$

 $cost_{mch} := c_{in}^{ed} + 0.25 \times c_{in}^{ed}(c_{imp} + c_{exp})$
 return $(\mathcal{G}_i^p,\ cost_{mch})$
```

Figure C.3: Algorithm: evaluate node matching cost

Algorithm import-edge-matching-cost ($\mathcal{G}_i^p$, n_k, i, S) =

Description:
This algorithm matches two groups of edges as a result of matching the pair of nodes $(n_{i,j}, n_k)$. Two groups of edges are: i) imported edge-bundles $\mathcal{R}_i^{mr^* \to pr_i}$ from some of $\mathcal{G}_u^{mr}$s to $\mathcal{G}_i^{pr}$; and ii) imported connector-edges $\mathcal{R}_i^{mr^* \to sr_i}$ from the same $\mathcal{G}_u^{mr}$s to $\mathcal{G}_{g(i)}^{sr}$. In this matching, the graph-edit cost is computed according to the imported connector-edge deletion/insertion costs defined in Section 6.4. The steps of algorithm are in Figure C.5.

Input:
$\mathcal{G}_i^p$, i: multi-phase tree-path (i.e., pattern-graph) at phase i.
n_k: node to be matched from search-space $\mathcal{G}_{g(i)}^{sr'}$.
S: system representation as the tuple $(G^s,\ D(N^s))$.

Output:
pattern-graph $\mathcal{G}_i^p$ (such that the connector-edges corresponding to the pair of nodes $(n_{i,j}, n_k)$ have been matched and updated in both pattern-region $\mathcal{G}_i^{pr}$ and link-module $\mathcal{G}_u^{mr}$) along with the cost of matching c_{imp}.

Local variables:
$\mathcal{R}_i^{mr^* \leftrightarrow pr_i}$: all connector-edges (i.e., edge-bundles and matched connector-edges) between every matched-region $\mathcal{G}_u^{mr}$ and pattern-region $\mathcal{G}_i^{pr}$.
$\mathcal{R}_i^{mr^* \leftrightarrow mr_u}$: all connector-edges (i.e., only matched connector-edges) between every matched-region and matched-region $\mathcal{G}_u^{mr}$.
$\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}$: group of imported edge-bundles from $\mathcal{G}_u^{mr}$ to $\mathcal{G}_i^{pr}$.
$\mathcal{R}_i^{mr_u \to pr_i|_{mch}}$: set of all imported matched connector-edges from $\mathcal{G}_u^{mr}$ to $\mathcal{G}_i^{pr}$.

r: number of edge-bundles in the group of all edge-bundles $\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}$.
u: id-number of link-module $\mathcal{G}_u^{mr}$.
min_u: minimum number of imported connector-edges to be matched.
N_u^{src} : set of all nodes in the link-module $\mathcal{G}_u^{mr}$ that are imported by node n_k but were not previously imported by $\mathcal{R}_i^{mr_u \to pr_i|_{mch}}$.
c_{imp}: cost of matching for imported connector-edges.

Continued in the next page ....

```
$(\mathcal{G}_i^{pr}, (\mathcal{R}_i^{mr^* \to pr_i}, \mathcal{R}_i^{mr^* \leftarrow pr_i})) := \mathcal{G}_i^p[i]$
$n_{i,j}$ := $current\text{-}placeholder\text{-}node(\mathcal{G}_i^{pr})$
$c_{imp} := 0.0$
% get import edges: from a link-module $\mathcal{G}_u^{mr}$ to pattern-region $\mathcal{G}_i^{pr}$.
for  $u \in [1 .. i-1]$  while  $c_{imp} <$ maxCost  do

   (if  $\mathcal{R}_i^{mr^* \to pr_i}[u] \neq$ NIL  then     % check if import link defined.
      $(\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}, \mathcal{R}_i^{mr_u \to pr_i|_{mch}}) := \mathcal{R}_i^{mr^* \to pr_i}[u]$
      $(\mathcal{G}_u^{mr}, (\mathcal{R}_i^{mr^* \to mr_u}, \mathcal{R}_i^{mr^* \leftarrow mr_u})) := \mathcal{G}_i^p[u]$
      $min_u := \epsilon^q((u, i, edge\text{-}type)).min$   % get min No. of import-edges.
      $r := |\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}|_{no.\ of\ bundles}$
      $N_u^{src} := get\text{-}new\text{-}source\text{-}nodes(n_k, N_u^{mr}, \mathcal{R}_i^{mr_u \to pr_i|_{mch}}, R^s)$
      % less than minimum number of import connector-edges are matched.
      (if  $j = |N_i^{pr}| \wedge |N_u^{src}| + |\mathcal{R}_i^{mr_u \to pr_i|_{mch}}|_{srcNodes} < min_u$  then
         $c_{imp} := maxCost$

      % more than maximum number of import connector-edges matched.
      else-if  $|N_u^{src}| > r$  then
         $c_{imp} := maxCost$

      % number of matched connector-edges (if any) is within the range.
      else
         $c_{imp} := c_{imp} + (1 - \frac{|N_u^{src}|}{r})$
         $\mathcal{R}_i^{mr_u \to pr_i|_{bdl}} := delete\text{-}edge\text{-}bundles(\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}, |N_u^{src}|)$
         $N_u^{src} := N_u^{src} \cup source\text{-}nodes(\mathcal{R}_i^{mr_u \to pr_i|_{mch}})$ % get all source nodes.
         for  $n_{src} \in N_u^{src}$  do
            if  $(n_{src}, n_k) \in R^s$  then
               $\mathcal{R}_i^{mr_u \to pr_i|_{mch}} := \mathcal{R}_i^{mr_u \to pr_i|_{mch}}$ with $(n_{src}, n_k)$

         $\mathcal{R}_i^{mr^* \to pr_i}[u] := (\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}, \mathcal{R}_i^{mr_u \to pr_i|_{mch}})$
         $\mathcal{R}_i^{mr^* \leftarrow mr_u}[i] := (\phi, \mathcal{R}_i^{mr_u \to pr_i|_{mch}})$
      )
      $\mathcal{G}_i^p[u] := (\mathcal{G}_u^{mr}, (\mathcal{R}_i^{mr^* \to mr_u}, \mathcal{R}_i^{mr^* \leftarrow mr_u}))$
   )
 $\mathcal{G}_i^p[i] := (\mathcal{G}_i^{pr}, (\mathcal{R}_i^{mr^* \to pr_i}, \mathcal{R}_i^{mr^* \leftarrow pr_i}))$
 return $(\mathcal{G}_i^p, c_{imp})$
```

Figure C.4: Algorithm: import edge matching cost

Steps of the algorithm: *import-edge-matching-cost*

(1) lines 5 to 12: Iterate in a loop and at each iteration consider one matched-region $\mathcal{G}_u^{mr}$ $(u \in [1 \,..\, i-1])$ that pattern-region $\mathcal{G}_i^{pr}$ has import-links from it (this is defined in AQL query text). At each iteration u perform the following operations: i) obtain the partially-matched imported connector-edges to $\mathcal{G}_i^{pr}$ (i.e., matched connector-edges $\mathcal{R}_i^{mr_u \to pr_i|_{mch}}$ and remaining edge-bundles $\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}$); ii) obtain $\mathcal{G}_u^{mr}$ and its corresponding matched connector-edges $\mathcal{R}_i^{mr^* \leftrightarrow mr_u}$ to be updated at the end of the loop; iii) obtain min_u to be used for constraint violation checking; iv) obtain the number of remaining edge-bundles r to be used for cost evaluation; and v) obtain the set of "non previously-imported" source nodes N_u^{src} (see Figure 7.3(b)) to be used for generating the matched connector-edges. The nodes N_u^{src} generate imported connector-edges that are used for min/max checking and cost evaluation.

(2) lines 14 to 15: Check for violating minimum number of imported connector-edges min_u. In this case, the current placeholder-node $n_{i,j}$ is the last placeholder-node to be matched with n_k, and the total number of imported nodes from $\mathcal{G}_u^{mr}$ is less than min_u. Therefore, n_k is rejected with maxCost.

(3) lines 18 to 19: Check for violating maximum number of imported connector-edges. Each candidate node n_k can potentially import between zero to all nodes of the link-module $\mathcal{G}_u^{mr}$. If the imported nodes in N_u^{src} are more than the remaining edge-bundles r, then node n_k is rejected with maxCost.

(4) lines 22 to 36: Match the imported connector-edges of n_k with the imported edge-bundle of $n_{i,j}$ as followings: i) evaluate the cost c_{imp} of deleting connector-edges from edge-bundle of $n_{i,j}$ (Section 6.4.2); ii) delete n edge-bundles from the remaining edge-bundles $\mathcal{R}_i^{mr_u \to pr_i|_{bdl}}$, where n is the number of imported fresh nodes. iii) generate all connector-edges (n_{src}, n_k) that n_k imports from $\mathcal{G}_u^{mr}$ (Note: n_{src} entitles both new and already imported nodes) and add the connector-edges to the set of matched connector-edges $\mathcal{R}_i^{mr_u \to pr_i|_{mch}}$; iv) restore the new partially-matched edge-bundles (i.e., new remaining edge-bundles and new matched connector-edges) to the group of all connector-edges $\mathcal{R}_i^{mr^* \leftrightarrow pr_i}$; and v) restore the link-module $\mathcal{G}_u^{mr}$ with its updated export connector-edges to the pattern-graph $\mathcal{G}_i^{p}$.

After all linked-modules $\mathcal{G}_u^{mr}$ of the pattern-region $\mathcal{G}_i^{pr}$ have been examined in the loop, the pattern-region and its updated partially-matched connector-edges are restored in the pattern-graph $\mathcal{G}_i^{p}$, which is returned along with the accumulated cost c_{imp}.

Figure C.5: Stepwise description of the algorithm *import-edge-matching-cost*

Algorithm export-edge-matching-cost ($\mathcal{G}_i^p$, n_k, i, S) =

Description:
This algorithm matches the exported edge-bundles $\mathcal{R}_i^{mr^* \leftarrow pr_i}$ (from $\mathcal{G}_i^{pr}$ to some of the $\mathcal{G}_u^{mr}$s defined in the AQL query) with the exported connector-edges $\mathcal{R}_i^{mr^* \leftarrow sr_i}$ (from $G_{g(i)}^{sr}$ to the same $\mathcal{G}_u^{mr}$s), that are caused by matching the pair of nodes $(n_{i,j}, n_k)$. The corresponding costs are defined in Section 6.4. This algorithm is similar to algorithm *import-edge-matching-cost*, and the only difference is that "at most one node (i.e., n_k) can be exported". This affects the checking for minimum/maximum number of connector-edges (lines 13 to 19) and the number of exported edges (lines 27 and 28).

Input:
$\mathcal{G}_i^p$, n_k, i: pattern-graph and node to be matched from $G_{g(i)}^{sr'}$, at phase i.
S: region representation of source-graph as tuple $(G^s, D(N^s))$.
Output:
pattern-graph $\mathcal{G}_i^p$ (such that the connector-edges corresponding to the pair of nodes $(n_{i,j}, n_k)$ have been matched and updated in both pattern-region $\mathcal{G}_i^{pr}$ and link-module $\mathcal{G}_u^{mr}$) along with the cost of matching c_{exp}.

Local variables:
$\mathcal{R}_i^{mr^* \leftrightarrow pr_i}$: all connector-edges (i.e., edge-bundles and matched connector-edges) between every matched-region $\mathcal{G}_u^{mr}$ and pattern-region $\mathcal{G}_i^{pr}$.
$\mathcal{R}_i^{mr^* \leftrightarrow mr_u}$: all connector-edges (i.e., only matched connector-edges) between every matched-region and matched-region $\mathcal{G}_u^{mr}$.
$\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}}$: group of exported edge-bundles from $\mathcal{G}_i^{pr}$ to $\mathcal{G}_u^{mr}$.
$\mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}}$: set of all exported matched connector-edges from $\mathcal{G}_i^{pr}$ to $\mathcal{G}_u^{mr}$.

r: number of edge-bundles in the group of all edge-bundles $\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}}$.
u: id-number of link-module $\mathcal{G}_u^{mr}$.
min_u: minimum number of exported connector-edges to be matched.
N_u^{sink} : set of all nodes in the link-module $\mathcal{G}_u^{mr}$ that import the node n_k.
c_{exp}: cost of matching for exported connector-edges.

Continued in the next page ....

```
$(\mathcal{G}_i^{pr},\ (\mathcal{R}_i^{mr^* \rightarrow pr_i},\ \mathcal{R}_i^{mr^* \leftarrow pr_i})) := \mathcal{G}_i^p[i]$
$n_{i,j}$ := $current\text{-}placeholder\text{-}node(\mathcal{G}_i^{pr})$
$c_{exp} := 0.0$
% get export edges: from pattern-region $\mathcal{G}_i^{pr}$ to a link-module $\mathcal{G}_u^{mr}$.
**for** $u \in [1 .. i-1]$ **while** $c_{exp} <$ maxCost **do**

    (**if** $\mathcal{R}_i^{mr^* \leftarrow pr_i}[u] \neq$ NIL **then**    % check if export link defined.
      $(\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}},\ \mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}}) := \mathcal{R}_i^{mr^* \leftarrow pr_i}[u]$
      $(\mathcal{G}_u^{mr},\ (\mathcal{R}_i^{mr^* \rightarrow mr_u},\ \mathcal{R}_i^{mr^* \leftarrow mr_u})) := \mathcal{G}_i^p[u]$
        $min_u := \epsilon^q((i, u, edge\text{-}type)).min$    % get min No. of export-edges.
        $r := |\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}}|_{no.\ of\ bundles}$
        $N_u^{sink} := get\text{-}sink\text{-}nodes(n_k,\ N_u^{mr},\ R^s)$
        % less than minimum number of export connector-edges are matched.
        (**if** $(|N_i^{pr}| - j) + |\mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}}|_{no.\ of\ srcNodes} < min_u$ **then**
          $c_{exp} := maxCost$

        % more than maximum number of export connector-edges matched.
        **else-if** $r = 0 \ \wedge\ |N_u^{sink}| > 0$ **then**
          $c_{exp} := maxCost$

        % number of matched connector-edges (if any) is within the range.
        **else**
          $c_{exp} := c_{exp} + 1$
          **if** $|N_u^{sink}| > 0$ **then**
            $c_{exp} := c_{exp} - 1$
            $\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}} := delete\text{-}edge\text{-}bundles(\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}}, 1)$
          **for** $n_{sink} \in N_u^{sink}$ **do**
            $\mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}} := \mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}}$ with $(n_k, n_{sink})$

          $\mathcal{R}_i^{mr^* \leftarrow pr_i}[u] := (\mathcal{R}_i^{mr_u \leftarrow pr_i|_{bdl}},\ \mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}})$
          $\mathcal{R}_i^{mr^* \rightarrow mr_u}[i] := (\phi,\ \mathcal{R}_i^{mr_u \leftarrow pr_i|_{mch}})$
        )
        $\mathcal{G}_i^p[u] := (\mathcal{G}_u^{mr},\ (\mathcal{R}_i^{mr^* \rightarrow mr_u},\ \mathcal{R}_i^{mr^* \leftarrow mr_u}))$
    )
  $\mathcal{G}_i^p[i] := (\mathcal{G}_i^{pr},\ (\mathcal{R}_i^{mr^* \rightarrow pr_i},\ \mathcal{R}_i^{mr^* \leftarrow pr_i}))$
  **return** $(\mathcal{G}_i^p, c_{exp})$
```

Figure C.6: Algorithm: export edge matching cost

Appendix D

AQL query example

This Appendix presents the AQL query for the Xfig system that is experimented in Chapter 8.

```
==================== AQL query for Xfig system ====================

BEGIN-AQL

  TYPE-OF-ANALYSIS:       SYSTEM = IO-F = SCORE-F2
  ANALYSIS-OR-DISTRIBUTE: ANA
  LINK-CONSTRAINTS:       YES
  ITEM-RELOCATION:        NO
  ITEM-DISTRIBUTION:      YES
  IO-SHRINK-OR-EXPAND:    G
  AUTO-OR-INCREMENTAL:    AUTO
  AUTO-ANALYSIS-SEQ:      FILE
  INCREMENTAL-STEPS:      FIRST: FILE: YES
  ANALYSIS-COMPONENTS:    S1, S2, S3, S4, S5
  MANUAL-COMPONENTS:
  MERGE-ANALYSIS-COMPS:   YES: S1: S4: RUN
  DECOMPOSE-SUBSYSTEM:    NO: S1: 4-MODULES: STOP
  COMPONENT-ASSOCIATION:  YES, REST-SYSTEM, 5-STEPS
  ASSOCIATION-VIEWS:      YES, F, T, V, FTV, 10-STEPS
  OUT--GRAPH-HTML:        YES, CODE, ss-LINK

SUBSYSTEM: S1-S4
   MAIN-SEEDS:    file u_elastic,
                  file u_drag
   SHRINK-EXPAND: E
   IMPORTS:
     RESOURCES:   rsrc ?IR,
                  rsrc ?R1(50 .. 150) S3
   EXPORTS:
     RESOURCES:   rsrc ?ER,
                  rsrc ?R2(40 .. 100) S3
   CONTAINS:
     FILES:       file $CL(4 .. 25),
                  file u_elastic,
                  file u_drag
   RELOCATES:     NO:
END-COMPONENT
```

```
SUBSYSTEM: S2
   MAIN-SEEDS:    file e_edit
   SHRINK-EXPAND: E
   IMPORTS:
     RESOURCES:   rsrc ?IR,
                  rsrc ?R3(40 .. 100) S3,
                  rsrc ?R4(0 .. 10) S5
   EXPORTS:
     RESOURCES:   rsrc ?ER
   CONTAINS:
     FILES:       file $CL(4 .. 20),
                  file e_edit
   RELOCATES:     NO:
END-COMPONENT

SUBSYSTEM: S3
   MAIN-SEEDS:    file e_scale
   SHRINK-EXPAND: E
   IMPORTS:
     RESOURCES:   rsrc ?IR,
                  rsrc ?R2(40 .. 100) S1-S4
   EXPORTS:
     RESOURCES:   rsrc ?ER,
                  rsrc ?R1(50 .. 150) S1-S4,
                  rsrc ?R3(40 .. 100) S2
   CONTAINS:
     FILES:       file $CL(4 .. 13),
                  file e_scale
   RELOCATES:     NO:
END-COMPONENT

SUBSYSTEM: S5
   MAIN-SEEDS:    file f_readtif
   SHRINK-EXPAND: E
   IMPORTS:
     RESOURCES:   rsrc ?IR
   EXPORTS:
     RESOURCES:   rsrc ?ER,
                  rsrc ?R4(0 .. 10) S2
   CONTAINS:
     FILES:       file $CL(4 .. 10),
                  file f_readtif
   RELOCATES:     NO:
END-COMPONENT
```

```
=================
REST-OF-SYSTEM:
=================
   IMPORTS:      S
   EXPORTS:      S
   CONTAINS:     E
     CLOSENESS:  L: 2
   DISTRIBUTES:
      files e_break, d_text, d_subspline, w_zoom, e_arrow TO: ALL;
      files e_align, w_rottext, e_update, w_layers, u_scale TO: ALL;
      files w_drawprim, u_redraw, u_create, w_modepanel, w_canvas TO: ALL;
      files f_util, mode, f_readxbm, w_cursor, w_setup TO: ALL;
      files f_readgif, u_print, u_pan, u_free, w_menuentry TO: ALL;
      files f_neuclrtab, f_readppm, u_error TO: ALL
END-COMPONENT
END-AQL
```

Appendix E

Glossary of terms

****** **A** ******

Alborz: a toolkit that implements the proposed architecture recovery and evaluation technique in this thesis (Section 8.2 Page 179).

AQL: Architecture Query Language is a textual language for composing a query specification that represents the architectural pattern of a software system (Section 4 Page 77, and Section 4.2 Page 80).

abstract component: a set of placeholders where the minimum/maximum cardinalities and the types of the placeholders are specified via the AQL query. An abstract component interacts with other abstract components through abstract connectors (Section 4.2 Page 80).

abstract connector: a set of placeholders that establish the interconnection among two abstract components. The minimum/maximum cardinalities and the types of the placeholders are specified via the AQL query (Section 4.2 Page 80).

abstract domain model: a schema to represent the software system entities and their interactions in an abstraction level that is suitable for an architectural recovery task (Section 3.1.1 Page 44).

architectural pattern: a set of partially specified components and a number of (size and type) constrained connectors among the components that collectively represent a macroscopic view of the core functionalities and interactions within the software system (Section 1.1 Page 4).

****** **B** ******

BQ-A*: Bounded queue A^* search algorithm proposed in this thesis, where the size of the sorted queue for the incomplete tree-paths has been restricted within a range, typically multiple hundreds of paths (Section 6.5 Page 142).

****** **C** ******

component: A named grouping of system entities (such as files, functions, datatypes, and variables) that imports and exports simple entities (such as functions, datatypes, and variables) from/to other groups of entities. A component is either a subsystem or a module (Section 4.1 Page 78).

composite entity: a file in the abstract domain model that contains a set of simple entities such as functions, datatypes, and variables (Section 3.1.1 Page 50).

connector: defined between two components as a group of simple entities that are defined in the source component and are used by the destination component (Section 1.1 Page 4).

connector-edges: a group of edges, denoted by $\mathcal{R}^{G_1 \leftrightarrow G_2}$, that connect two graphs G_1 and G_2 in uni-directional (using $\leftarrow$ or $\rightarrow$) or bidirectional (using $\leftrightarrow$) mode (Section 5.2.4 Page 108).

cont-R: containment relation between a file and a simple entity in the abstract domain model, such that each simple entity can be contained in only one file (Section 3.1.1 Page 50).

****** **D** ******

domain of a node (entity) D^{n_j}: a set of source-graph nodes (along with their similarity values to n_j) that are associated with node n_j, where n_j is called the main-seed of the domain (Section 3.3.2 Page 71).

****** **E** ******

edge-bundle: a group of edges with a specific type that connect every node in a matched-region G_u^{mr} to one node (either a common sink-node or a common source-node) in the pattern-region G_i^{pr} such that the types of the nodes conform with the type of edge-bundle and $u < i$. (Section 5.2.4 Page 107).

entity association: similarity measure between two simple entities e_i and e_j based on the maximal association among a group of simple entities including e_i and e_j, and denoted as $entAssoc(e_i, e_j)$ (Section 3.3 Page 64).

environment: a set of techniques along with a process that collectively perform a specific task (Section 1.6 Page 15, and Section 8.2 Page 179).

export: a component (or a file) exports all its contained simple entities that are used by the simple entities that it does not contain (Section 4.1 Page 78, and page 94).

exp-R: relation export that is defined between a file and a simple entity (resource) in the abstract domain model (Section 3.1.1 Page 51).

****** **F** ******

file-level: an abstraction level where "file" is the highest granularity level of the system entities (Section 3.1.1 Page 50).

frequent itemset: in data mining algorithms, a set of items that is contained in every basket of a group of baskets (called supporting transactions). The cardinality of this group of baskets must be greater than a user-defined threshold called minimum support (Section 3.2.2 Page 60).

function-level: an abstraction level where "function" is the highest granularity level of the system entities (Section 3.1.1 Page 49).

****** **G** ******

graph summation: the notations "plus $+$" for connecting two graphs, and "oplus $\oplus$" for connecting a graph to a group of connector-edges are used to model the pattern matching process as graph algebraic equations (Section 5.2.4 Page 109).

group association: similarity measure between two groups of entities g_i and g_j such as two files, based on the "entity association" similarity between every pair of entities one entity in each group, denoted as $groupAssoc(g_i, g_j)$ (Section 3.4 Page 72).

****** **I** ******

import: a component (or a file) imports all the simple entities that are used by its contained simple entities, but are not contained by that component (file) (Section 4.1 Page 78, and Page 93).

imp-R: relation import that is defined between a file and a simple entity (resource) in the abstract domain model (Section 3.1.1 Page 51).

input-graph G_i^I: a sub-graph of the source-graph G^s. The input-graph and pattern-graph G_i^p are supplied to the matching process at phase i to be matched and produce the matched-graph G_i^m (Section 5.2.7 Page 115).

itemset: refer to "frequent itemset" in this glossary.

****** **M** ******

main-seed: refer to "domain of a node" in the glossary.

matching phase: the whole pattern matching process is divided into k incremental phases, where k is the number of AQL query abstract-components and the current matching phase is denoted by "i" (Section 6.1.3 Page 121).

matched-graph G_i^m: the result of the matching process applied on two graphs, pattern-graph G_i^p and input-graph G_i^I. (Section 5.2.5 Page 109).

matched-region G_u^{mr}: the result of matching a pattern-region G_i^{pr} with a source-region $G_{g(i)}^{sr}$ at matching phase i (Section 5.2.3 Page 107).

maximal association: defined in a group of entities in the form of a maximal set of entities that all share the same relation to every member of another maximal set of entities (Section 3.2 Page 55).

minimum support: refer to "frequent itemset" in this glossary.

module: a component that contains simple entities and imports/exports simple entities (Section 4.1 Page 78, and Page 99).

****** **P** ******

pattern: refer to "architectural pattern" in this glossary.

pattern-graph G_i^p: generated by incrementally expanding the query-graph G^q at different matching phases. At matching phase i: i) the i_{th} node of the query-graph is expanded into a pattern-region G_i^{pr}; and ii) each edge of the query-graph between the i_{th} node and the already expanded nodes are expanded into edge-bundles (Section 5.2.6 Page 111).

pattern-region G_i^{pr}: generated by expanding the i_{th} node of the query-graph G^q (qn_i) at matching phase i, through: i) generating maximum number of placeholder nodes defined by qn_i; and ii) connecting every node in the pattern-region to every other node that is allowed based on the types of the nodes (Section 5.2.2 Page 106).

placeholder: a node in the pattern-region G_i^{pr} that can be matched with a system entity in the source-region $G_{g(i)}^{sr}$ during the matching process (Section 4.2 Page 80).

****** **Q** ******

query-edge: an edge of the query-graph G^q (Section 5.2.1 Page 105).

query-graph G^q: a multi-graph with composite nodes (denoted as query-nodes) and composite edges (denoted as query-edges) that describes a macroscopic pattern of the system components and their interaction constraints (Section 5.2.1 Page 105).

query-node: a node of the query-graph (Section 5.2.1 Page 105).

****** **S** ******

simple entity: a function, aggregate/array datatype, or global variable, defined in the abstract domain model, that is contained in a file as a composite entity (Section 3.1.1 Page 50).

software architecture: a partition of the software system entities into cohesive components that reflect the system characteristics and domain knowledge, and meets the structural constraints defined by a given architectural pattern (Section 1.1 Page 3).

software architecture recovery: refers to the problem of devising a tractable process, required techniques, and the supporting tools for interactively and incrementally extracting a system's structure using domain and system knowledge (Section 1.2 Page 4, and Section 1.4 Page 8).

source-level domain model: a schema to represent the software system entities and their interactions at the source-code level that are defined by the corresponding programming language constructs (Appendix A.1 Page 231, and Section 3.1.1 Page 45).

source-graph G^s: graph representation of a software system for architectural analysis that is obtained using the proposed abstract domain model (Section 3.1.2 Page 51).

subsystem: a component that contains files (composite entities) and their contained simple entities, and imports/exports simple entities (Section 4.1 Page 78, and Page 90).

****** **U** ******

use-F: relation between two functions in the abstract domain model whose implementations at the source-level domain model are related by the call relation (Section 3.1.1 Page 49).

use-R: a relation between a file and a simple entity in the abstract domain model that corresponds to a relation between a file and a simple entity in the source-level domain model, where the file contains a function and the function calls/reads/updates the simple entity (Section 3.1.1 Page 50).

use-T: a relation between a function and a datatype in the abstract domain model, where the implementation of the function in the source-level domain model reads/updates a variable, and the variable is of an aggregate/array type (Section 3.1.1 Page 49).

use-V: a relation between a function and a variable in the abstract domain model whose implementations in the source-level domain model are related by the read/update relation (Section 3.1.1 Page 49).

****** **V** ******

views: views are the result of applying separation of concerns on a development or recovery process in order to classify the related knowledge about that process into more understandable and manageable forms (Section 2.2 Page 27).

Bibliography

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

[2] PBS: The Portable Bookshelf,
URL = http://www.turing.toronto.edu/pbs/.

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

[4] CLIPS expert system builder,
URL = http://www.ghg.net/clips/CLIPS.html.

[5] Bash Unix shell, URL = http://www.delorie.com/gnu/docs/bash/.

[6] Apache HTTP Server, URL = http://www.apache.org/httpd.html.

[7] XML: Extensible Markup Language, URL = http://www.w3.org/XML/.

[8] ELM Pages,
URL = http://www.math.fu-berlin.de/ guckes/elm/elm.index.html.

[9] Ghostview postscript file viewer,
URL =http://wwwthep.physik.uni-mainz.de/ plass/gv/.

[10] SEI Software Architecture Definitions,
URL = http://www.sei.cmu.edu/architecture/definitions.html.

[11] Rakesh Agrawal, K. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 490–501, 1995.

[12] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 487–499, 1994.

[13] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[14] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[15] Nicolas Anquetil and Timothy C. Lethbridge. Experiments with clustering as a software remodularization. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, 1999.

[16] Denis Avrilionis and Pierre-Yves Cunin. View-based mechanisms for structured and distributed enactment. In *Proceedings of the International Work-*

shop on Multiple Pespectives in Softwre Development (viewpoints '96), pages 259–262, 1996.

[17] Sidney Bailin. Kaptur: a tool for the preservation and use of engineering legacy. Unpublished paper, copyright Sidney Bailin, 1992.

[18] Victor R. Basili and Richard W. Selby. Paradigm for experimentation and empirical studies in software engineering. *Reliability Engineering and System Safety*, 32(1-2):171–191, 1991.

[19] Bell, IBM. *Workgroup on Standard Exchange Format (WoSEF)*, Limerick, Ireland, June 06 2000.

[20] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-specific software architectures for guidance, navigation and control. Technical report, Honeywell Technology Center, 1994.

[21] Garrett Birkhoff. *Lattice Theory.* American Mathematical Society, 1st edition, 1940.

[22] Michael Blaha. A retrospective on industrial database reverse engineering projects- parts 1 and 2. In *Proceedings of the Working Conference on Reverse Engineering*, pages 136–153, 2001.

[23] Ivan Bowman, Michael W. Godfrey, and Ric Holt. Connecting architecture reconstruction frameworks. *Journal of Information and Software Technology*, 42(2):93–104, February 2000.

[24] Ivan T. Bowman, R.C. Holt, and N.V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the ICSE'99*, pages 555–563, 1999.

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

[26] Gerardo Canfora, Jorg Czeranski, and Rainer Koschke. Revisiting the delta ic approach to component recovery. In *Proceedings of the WCRE'00*, pages 140–149, 2000.

[27] David N. Chin and Alex Quilici. Decode: A co-operative program understanding environment. *Software Maintenance: Research and Practice*, 8:3–33, 1996.

[28] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *IEEE 17'th International Conference on Software Engineering (ICSE)*, pages 73–82, 1995.

[29] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[30] John Davey and Elizabeth Burd. Evaluating the suitability of data clustering for software remodularisation. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 268–276, 2000.

[31] Carlos Montes de Oca and Doris L. Carver. A visual representation model for

software subsystem decomposition. In *Proceedings of the Working Conference on Reverse Engineering*, pages 231–240, 1998.

[32] Thomas R. Dean and James R. Cordy. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313, 1995.

[33] Prem. T. Devanbu. Genoa - a customizable, language and front_end independent code analyzer. In *Proceedings of the 14th ICSE*, pages 307–317, 1992.

[34] 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.

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

[36] Brian S. Everitt. *Cluster Analysis.* John Wiley, 1993.

[37] Usama M. Fayyad. *Advances in knowledge discovery and data mining.* MIT Press, Menlo Park, Calif., 1996.

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

[39] R. Fiutem, E. Merlo, G. Antoniol, and P. Tonella. Understanding the architecture of software systems. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 187–196, 1996.

[40] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliche-based environment to support architectural reverse engineering. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 319–328, 1996.

[41] L. R. Foulds. *Graph Theory Applications.* Springer-Verlag, 1991.

[42] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm. On the definition of software system architecture. In *ICSE 17 Software Architecture Workshop*, April 1995.

[43] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, 1997.

[44] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.

[45] Jean-Francois Girard and Rainer Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 58-65 1997.

[46] David A. Grossman and Ophir Frieder. *Information Retrieval: algorithms and Heuristics.* Kluwer Academic Publishers, 1998.

[47] John Guttag and James J. Horning. *Larch : languages and tools for formal specification.* Springer Verlag, 1993.

[48] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th ICSE*, pages 186–195, 1995.

[49] R.C. Holt, A. Winter, and A. Schurr. Gxl: Toward a standard exchange format. In *Proceedings of the Working Conference on Reverse Engineering*, pages 162–171, 2000.

[50] Richard C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering*, 1998.

[51] Gerard Huet. From an informal textual lexicon to a well-structured lexical database: an experiment in data reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering*, pages 127–135, 2001.

[52] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

[53] Wilfried Imrich and Sandi Klavzar. *Product Graphs: Structure and Recognition.* John Wiley, 2000.

[54] Anil K. Jain. *Algorithms for Clustering Data.* Prentice Hall, Englewood Cliffs, N.J., 1988.

[55] Rick Kazman and Marcus Burth. Assessing architectural complexity. In *Proceedings of the CSMR*, pages 104–112, 1998.

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

[57] Rick Kazman, Paul Clements, Gregory Abowd, and Len Bass. Classifying architectural elements as a foundation for mechanism matching. In *Proceedings of the COMPSAC*, pages 14–17, 1997.

[58] Kostas Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for design concept localization. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'95)*, pages 96–103, 1995.

[59] Rainer Koschke. An incremental semi-automatic method for component recovery. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 256–267, 1999.

[60] Rainer Koschke and Thomas Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the IWPC*, pages 201–210, 2000.

[61] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[62] Thomas Kunz and James P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527, June 1995.

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

[64] Arun Lakhotia and John Gravley. Toward experimental evaluation of sybsystem classification recovery techniques. In *Proceedings of the WCRE*, pages 262–269, 1995.

[65] K. Lano. *Formal Object-Oriented Development.* Springer, 1995.

[66] Kurt Lichtner, Paulo Alencar, and Don Cowan. Using view-based models to formalize architecture description. In *Proceedings of the Third International Software Architcture Workshop, ISAW-3*, 1998.

[67] Kurt Lichtner, Paulo Alencar, and Don Cowan. An extensible model of architecture description. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 156–165, 2000.

[68] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, 1997.

[69] P. Linos, L. Aubet, Y. Dumas, P. Helleboid, and P. Tulula. Visualizing program dependencies. *Journal of Software-Practice and Experience*, 24(4):387–403, 1994.

[70] David C. Luckham, John J. Kenny, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture

using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[71] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[72] 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 the IWPC*, pages 45–53, 1998.

[73] James Martin and Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[74] Nabor C. Mendonca and Jeff Kramer. Requirements for an effective architecture recovery framework. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 101–105, 1996.

[75] Bruno 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.

[76] Brian S. Mitchell and Spiros Mancoridis. Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of the WCRE*, pages 93–102, 2001.

[77] Robert T. Monroe. Capturing design expertise in software architecture design

environments. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 87–89, 1996.

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

[79] 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.

[80] Jeffery S. Poulin. Evolution of a software architecture for management information systems. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 134–137, 1996.

[81] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[82] James M. Purtilo. The polylith software bus. Technical Report UMIACS-TR-90-65, CS-TR-2469, Department of Computer Science, University of Maryland, College Park, MD 20742, May 1990.

[83] A. Quilici and D. N. Chin. Decode: A cooperative environment for reverse-engineering legacy software. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 156–165, 1995.

[84] Reasoning Inc., 700 E. El Camino Real, Mountain View, CA 94040, USA.

Software Development Kit: Refine/C User's Guide for Version 1.2, April 1998.

[85] Reasoning Systems Inc., Palo Alto, CA. *Refine User's Guide*, version 3.0 edition, May 1990.

[86] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[87] Kamran 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.

[88] Kamran 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.

[89] Kamran Sartipi and Kostas Kontogiannis. Component clustering based on maximal association. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 103–114, Stuttgart, Germany, October 2001.

[90] Kamran Sartipi and Kostas Kontogiannis. Interactive software architecture recovery: An incremental supervised clustering approach. Technical Report UW-E&CE#2002-06, Dept. E&CE, University of Waterloo, Waterloo, Canada, April 2002.

[91] Kamran Sartipi and Kostas Kontogiannis. A user-assisted approach to component clustering. *Accepted for the Journal of Software Maintenance: Research and Practice (JSM)*, 2002.

[92] Kamran Sartipi, Kostas Kontogiannis, and Farhad Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of the IEEE IWPC*, pages 37–47, Limerick, Ireland, June 2000.

[93] Thomas Schiex, Helene Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the IJCAI-95*, pages 631–637, 1995.

[94] Artificial Intelligence Section. *CLIPS Architectural Manual Version 4.3*. Lyndon B. Johnson Space Center, jsc-23047 edition, May 1989.

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

[96] Mary Shaw, Robert DeLine, et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[97] Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, 1995.

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

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

[100] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering*, pages 196–207, 1995.

[101] Ryan D. Stansifer. *The Study of Programming Languages.* Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[102] Janos Sztipanovits et al. Multigraph: An architecture for model-integrated computing. In *Proceedings of the IEEE ICECCS'95*, pages 361–368, 1995.

[103] Allan Terry et al. An annotated repository schema, domain-specific software architecture. Technical report, TFS and ARDEC, October 1993.

[104] S. R. Tilley, H. A. Muller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Proceedings of the International Conference on Software Maintenance*, pages 142–151, 1993.

[105] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *Proceedings of the International Conference on Software Maintenance (ICSM'01)*, pages 398–407, 2001.

[106] Vassilios Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

[107] Vassilios Tzerpos and Richard C. Holt. Software botryology: Automatic clustering of software systems. In *Proceedings of the International Workshop on Large-Scale Software Composition*, pages 811–818, 1998.

[108] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.

[109] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 246–255, 1999.

[110] Ernest Wallmuller. *Software Quality Assurance: A Practical Approach.* Prentice Hall, New York, 1994.

[111] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, 1996. Page 19.

[112] T. A. Wiggerts. Using clustering algorithms in legacy systems modularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43, 1997.

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

[114] Steven G. Woods and Qiang Yang. Program understanding as constraint satisfaction. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 314–323, 1995.

[115] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

[116] Jingwei Wu and Margaret-Anne D. Storey. A multi-perspective software visualization environment. In *Proceedings of the CASCON conference*, pages 41–50, 2000.

[117] John. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292, 1987.