Book title:

Managing Corporate Information Systems Evolution and Maintenance

Chapter title:

Software Architecture Analysis and Reconstruction

Kamran Sartipi

Assistant Professor

Department of Computing and Software, McMaster University

Hamilton, ON, L8S 4K1 Canada

Tel: +1-905-525-9140 (ext. 26346)

Fax: +1-905-524-0340

Email: sartipi@cas.mcmaster.ca

&

Kostas Kontogiannis

Associate Professor

Department of Electrical & Computer Engineering

University of Waterloo

Waterloo, ON, N2L 3G1 Canada

Tel: +1-519-888-4567 (ext. 2840)

Fax: +1-519-746-3077

Email: kostas@swen.uwaterloo.ca

Software Architecture Analysis and Reconstruction

ABSTRACT

Software architecture analysis and reconstruction is an area within the software architecture domain that refers to the techniques and tools for processing and recovering high-level information from a lower-level system representation such as the source code. The importance of architecture analysis and reconstruction of a large and complex software system stems from the need to perform continuous maintenance activities to keep a mission critical system operational. Such activities include, adopting a new implementation technology, error correction, feature enhancement, and migration to a new platform, where the architectural reconstruction constitutes the major part of all of these activities. The approaches to architectural reconstruction are expected to address the following issues: specific views of the system to extract; representation models for the software system entities and relations; high-level models of the software system to reconstruct; architecture reconstruction techniques; tractability of the reconstruction process; and evaluation methods for the result of the reconstruction process. In this Chapter, first, the research challenges of this field are presented, and second, the state-of-the-art and practice solutions to these challenges are discussed.

INTRODUCTION

For several decades, we have been witnessing the importance and influence of large and complex software systems into various aspects of our lives. In recent years the engineers have been confronted with the problem of *legacy systems* which are large, mission critical, and complex software systems that have been operational and modified over a number of years. These old systems are difficult and costly to maintain, evolve, or integrate with other systems since they usually lack any updated design documents and possess complex structures. The average life-time of legacy software systems is between 10 to 15 years (Wallmuller, 1994) and the replacement of these systems is very expensive, therefore these systems are subject to re-designing and re-engineering.

Software architecture analysis and reconstruction encompasses various methods and supporting tools for extracting high-level information from some lower level representation of a software system such as the source code. Architectural reconstruction is a reverse-engineering activity that focuses on the architectural design aspects. In all of the following software maintenance activities, the software analysis and architectural reconstruction are the major parts of the operations on legacy software systems. Adopting a new technology such as, object-orientation, component-base programming, or network-centric re-engineering requires changes in the design of the system, hence the design of the system must be well understood before such activities commence. Error-correction and feature enhancement operations invalidate the design document of the system, therefore the design document must be updated. Migrating a legacy system to a new platform

such as Windows or Unix operating system requires functional and behavior description of the system's components, which requires to understand the components functionality and behavior.

The discussions in this Chapter are geared towards two main objectives. First, to cover the definitions and the major issues in the field of software architecture reconstruction and the challenges that the researchers in this field encounter. These issues include: specific views of the system to extract; representation models for the software system entities and relations; architecture reconstruction techniques; tractability of the reconstruction process; and evaluation methods for the result of the reconstruction process. Second, to address the possible solutions to these issues as they are presented by the state-of-the-art techniques in this field. These techniques include: clustering techniques, concept-lattice analysis, query-based techniques, and system composition and visualization. Finally, the proposed future trends in this Chapter will serve as a starting point for further research in this field.

SOFTWARE ARCHITECTURE RECONSTRUCTION

The following observations form the basis for a definition for software architecture reconstruction. Despite several attempts for automating the architectural reconstruction process 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 reconstruction should be an incremental process. Software systems usually consist of architectural patterns in their design which form the basis for the reconstruction process.

Most reconstruction 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 reconstruction process. Based on the above discussion, in (Sartipi, 2003) the software architectural reconstruction is defined as:

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

The whole software reconstruction 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 process constructs a high-level view of the system from the source model.

For the discussions in this Chapter, the software architecture, component, connector, and architectural pattern are defined as follows:

- *Software architecture*: a partition of the software system entities into components that reflects 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) according to some architectural properties, that interacts with other groups through using or providing the system services.

- *Connector*: a relation abstraction between two components using a group of system entities (e.g., files, functions, datatypes, and variables) that allow the interaction between two components.

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

Issues in Software Architecture Reconstruction

This Section provides a brief overview on various issues that an approach to software architecture reconstruction must address. A detailed discussion on each issue will be presented in a corresponding Section.

What 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. Section "Architectural views" introduces a set of views of a software that is suitable for reconstruction process.

How to Represent the Software System. In software architecture reconstruction 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. In Section "Software system representation" the methods for representing low-level software system using the notion of domain model are discussed.

What Reconstruction Technique to Use. In a nutshell, approaches to architectural reconstruction 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. The pattern-based approaches first compose a high-level mental model of the system architecture (i.e., the conceptual architecture or architectural pattern) using a modeling means, and a search engine identifies the pattern in the software representation. The clustering category and pattern-based category are further divided into several techniques each with particular advantages and disadvantages that make the basis for the user to adopt a technique. In Section "Techniques for architectural reconstruction" different techniques will be presented.

How to Make the Reconstruction Process Tractable. 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 reconstruction tasks. In Section "Scalability of the reconstruction process"

several heuristics to deal with scalability will be discussed.

How to Involve the User in Reconstruction. The role of the user, as an integral part of an architectural reconstruction process, is important for directing the reconstruction process. In fact, the ambitious goal of fully automating the reconstruction 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 reconstruction complexity. This trend necessitates that the domain knowledge and system documents be incorporated in the reconstruction process by the user inspection. The Section "User involvement in the reconstruction process" addresses the importance of user in this process.

How to Validate the Recovered Architecture. Similar to validation testing in forward engineering, a reverse engineering process is also expected to generate results that can be validated against the actual or intended 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 Section "Architectural evaluation techniques" the current techniques for assessing the result of the reconstruction process are discussed.

In the rest of this Chapter, the detailed discussions on the above issues along with the proposed solutions by the different techniques are presented.

ARCHITECTURAL VIEWS

The significance of software architecture *views* has been widely addressed in the literature (Zachman, 1987; Kruchten, 1995; Soni, Nord, & Hofmeister, 1995; Poulin, 1996). In a broad sense, views are the result of applying *separation of concerns* on a development or reverse engineering process in order to classify the related knowledge about that process into more understandable and manageable forms. The choice of an appropriate set of views is a common concern both in software development and reconstruction process. The proposed sets of views for developing or specifying a system consists of: data, function, and network (Zachman, 1987); function, process, development, and physical (Kruchten, 1995); conceptual, module-interconnection, execution, and code (Soni et al., 1995).

In general, it is ideal to recover the same set of views of a system that is also needed for its development. Unfortunately, reverse engineering is much more difficult than forward engineering. Figure 1 illustrates a categorization of essential features used for describing software architecture and a set of three architectural views, namely *structure, behavior* and *environment* that are suitable for reconstruction. The chosen views are orthogonal and carry most of the important information that encompass the systems in different domains such as information, concurrent, reactive, distributed, and command and control. A brief description of the different views follows.

Structure View

The structural view covers all building blocks and interconnections (glues) that statically describe the architecture of a software system.

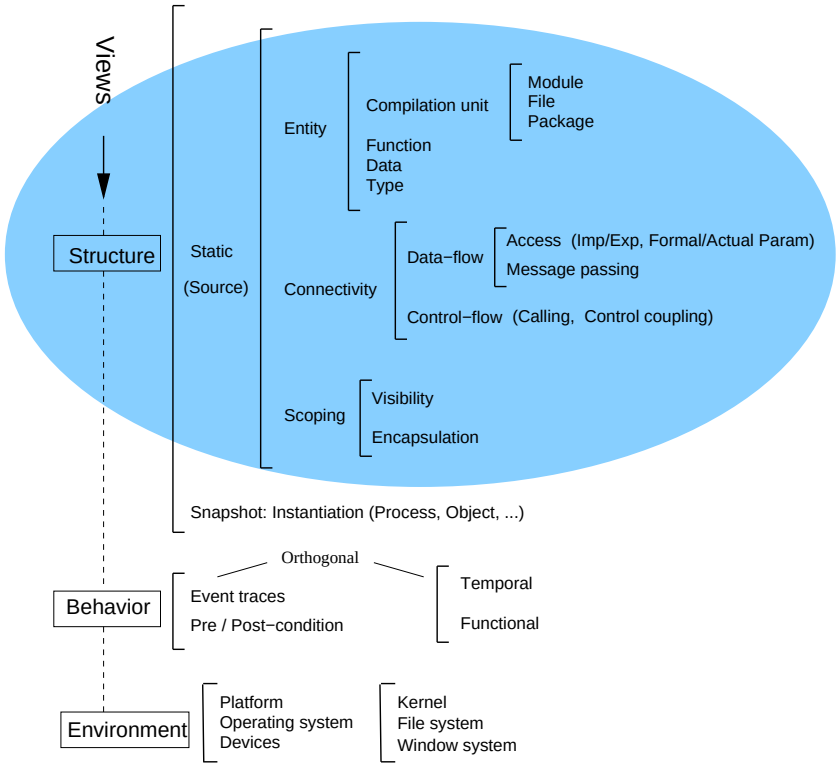

Figure 1. Classification of a software system features into three views that are intended for architectural reconstruction purposes. The grey area highlights the scope of architecture reconstruction techniques discussed in this Chapter.

The structural view is the most appropriate architectural view to be recovered and a number of approaches and tools already exist (Finnigan et al., 1997; 31,). The structural view consists of two parts:

a) The *static* features are the property of the source code, hence, can be extracted by static analyzing the source program. An *entity* refers to a basic block of code that constitutes in building a software's structure. A *connectivity* refers to an interconnection between two entities. A *scope* refers to the maximum range that a definition is effective. Scope is further divided into *visibility* such as local and global; and *encapsulation* such as

public and private access privileges.

b) The *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 information to be discovered.

Behavioral View

The behavioral view of a system refers to the services that a system provides to its environment through its interactions with the environment.

The behavioral view can be expressed from two orthogonal aspects of the program properties such as: i) event traces (sequences of function invocations using the profiling techniques) (Bojic & Velasevic, 2000; El-Ramly, Stroulia, & Sorenson, 2002; Eisenbarth, Koschke, & Simon, 2001) and Pre/Post-conditions (input/output constraints of a function or module) (Zaremski & Wing, 1995); ii) temporal properties (concurrency issues) and functional properties (data transformation characteristics) which are difficult to be analyzed and recovered.

Environment View

The environment view of a software system (application) refers to all supporting facilities, including hardware and software, that encompass the system and enable it to operate and provide its services to the environment.

The environment view consists of: platform, operating system, and devices, where the operating system can be further subdivided.

Specifically, the grey area in Figure 1 highlights the scope of the architecture reconstruction techniques discussed in this Chapter. The techniques pertinent to the recovery of different views of a system are significantly different in both the system representation and the adopted reconstruction technique. Therefore, as an important design decision to make, the engineer should restrict the scope of the reconstruction to a subset of the features in a view of the system. For example, the highlighted area in Figure 1 except the features "Scoping, Message passing, and Control coupling" indicates the scope of a pattern-based architectural reconstruction approach presented in (Sartipi, 2003).

SOFTWARE SYSTEM REPRESENTATION

In software architecture reconstruction, an appropriate choice of a source model is central in: recovering desired properties from a program; matching algorithm efficiency; and programming language independence. In this Section, the alternative models for representing the parsed software system are introduced. Because of its expressiveness, the graph representation of a software system has been adopted by most of the current approaches, however, different graph modeling techniques can be used.

Domain Model

In a software system, entities and their interactions both at design level and source-code level can be represented by various formalisms, including: entity relation diagrams, module interconnection diagrams, structure charts, program dependency graphs, and ab-

stract syntax graphs. A domain model provides a schema for such a representation formalism and can be represented as a class diagram. For example, the entity relation diagram of a system at the source-code level can be extracted from the domain model of the corresponding programming language, by considering: i) source code constructs as instantiations of domain model classes (i.e., file, function, statement, expression, type-specifier, and variable); and ii) relationships between source code entities as instantiations of associations between domain model classes. The instantiation of the classes and associations into objects and relationships is the result of parsing a software system according to the corresponding grammar where an abstract syntax tree of the class objects and their links is generated. Figure 2 illustrates a simplified class diagram of the domain model for a typical procedural language. This model has been derived from a complete domain model for the C language (refineC98, 1998), and can be used as the basis to define a source model for the software system.

Source Model Definition. The source model for an architecture reconstruction process is a database of entities and relationships that is used by the analysis process in order to build a high-level view of the system, i.e., software architecture. The source model is defined via a domain model that is derived from the source-level domain model and defines the types of entities and relationships that are suitable for a specific architecture-level analysis. The resulting domain model deletes the unnecessary details of entities and relationships and hence it is sometimes called *abstract domain model* (Sartipi & Kontogiannis, 2003), *concept model* (Chen, Nishimoto, & Ramamoorthy, 1990), or *schema*

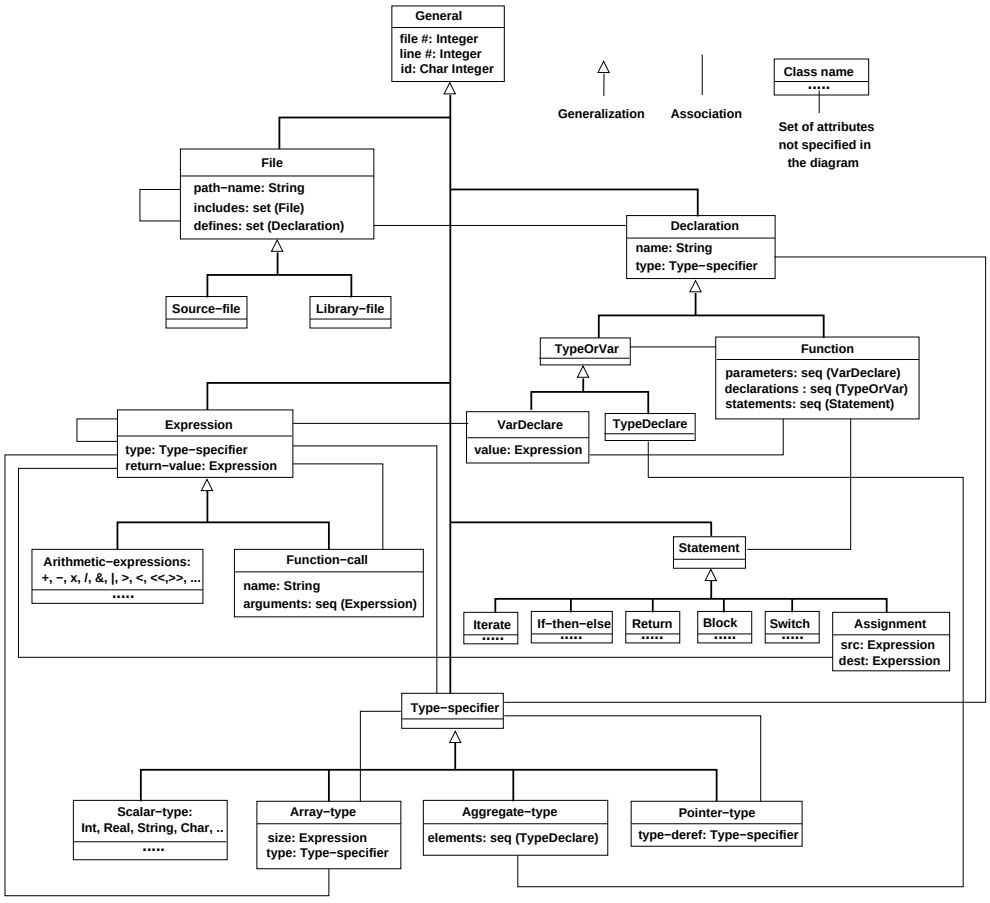

Figure 2. The class diagram of a simplified domain model for a typical procedural programming language such as C. The instantiation of the classes and their association links during the parsing process generates an abstract syntax tree for the software system.

(Finnigan et al., 1997). In such a domain model, the set of entity-types is a sub-set of entity-types in the source-level domain model and each relation-type is an aggregation of one or more relation types in the source-level domain model. The source models of four reconstruction environments are discussed below.

The Alborz environment (Sartipi, 2001a) defines an "abstract domain model" in which the types of entities are: source file, function, datatype (including aggregate and array types), and global variables. Also, the types of relations are: use-F, use-T, use-V,

Entities	
Source code entities	**Architectural entities**
source-file "*main.c*"	abstract-file L_i
function "*foo*"	abstract-function F_j
aggregate-type "*bar*" or array-type "*bar*"	abstract-type T_k
global-variable "*kam*"	abstract-variable V_m

Relations	
Source code relations	**Architectural relations**
function "*foo*" calls function "*foobar*"	F_j *use-F* F_x
function "*foo*" passes, receives, or uses aggregate-type / array-type "*bar*"	F_j *use-T* T_k
function "*foo*" references or updates global-variable "*kam*"	F_j *use-V* V_m
source-file "*main.c*" defines function "*foo*", defines aggregate-type / array-type "*bar*", defines global-variable "*kam*"	L_i *cont-R* F_j L_i *cont-R* T_k L_i *cont-R* V_m
one or more functions defined in source-file "*main.c*" call function "*foo*", or use aggregate-type / array-type "*bar*", or reference/update global-variable "*kam*"	L_i *use-R* F_j L_i *use-R* T_k L_i *use-R* V_m

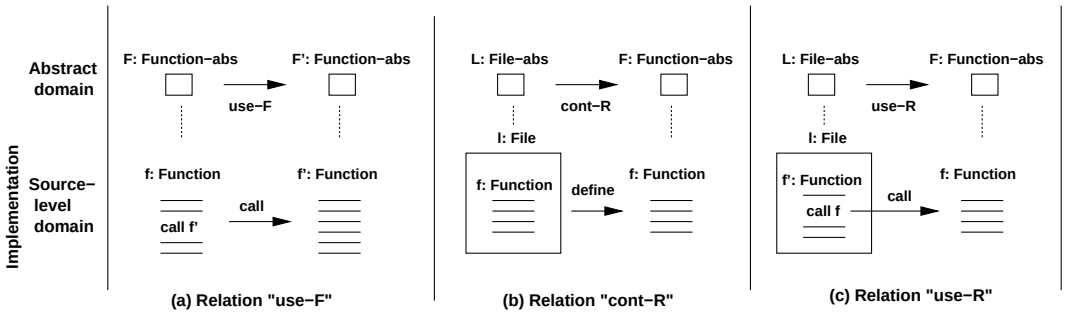

Figure 3. The relationships between the types of entities and relations in a procedural language source code, such as C, with the types of entities and relationships at the architectural level. L_i, F_j, T_k, V_m are unique entity identifiers. Each relation at the architectural level is an aggregation of one or more relations in the source code. At the bottom, the abstraction of three composite relations in the source-level domain model, using the abstract domain model are shown.

cont-R, use-R, imp-R, and exp-R. Figure 3 illustrates the correspondence between the types of entities and relations in the source code (defined by the source-level domain model) and the types of entities and relationships at the architectural level (defined by the abstract domain model in (Sartipi & Kontogiannis, 2003)). This source model allows to analyze a software system at two levels of entity granularity, i.e., at module-level the lower grained entities such as functions, datatypes and variable are analyzed, and at subsystem-level the higher grained entities such as files are analyzed.

Now, we discuss how the source-level domain model in Figure 2 can be used to specify the steps for extracting the entities that are related according to the aggregated relations in the abstract domain model we defined above. As a simple example, we extract all the functions that are related with function "foo" with the relation *use-F* which is in fact the simple relation *call function*. The Steps are as follows:

1. Get the sequence S of statements in function "foo"

2. For each statement s in S that is either: *iteration, if-then-else, return, block, switch, assignment*, get all the expressions E in s.

3. Check each expression e in E and keep only those that are of type *Function-call*. The name of the called function can be obtained from the "name" attribute of the function-call expression.

In the case of an aggregate abstract relation such as *use-V*, each simple relation such as reading from the global variable and writing to the global variable must be extracted separately, and consequently their relations be aggregated.

The CIA (C Information Abstraction System) (Chen et al., 1990) defines a *concept model* using an attributed entity relation diagram where the entities represent "file, function, type, global variable, and macro" and the relations represent the "reference" relation. The attributes for an entity define the name and location properties of the entity. In this model a "reference" relation is defined between two entities A and B if the definition of A refers to B such that A can not be compiled and executed without the definition of B. Therefore, in this model the relation "reference" denotes to either of the relations in the above abstract domain model.

The PBS (Portable BookShelf) (Finnigan et al., 1997) uses a complex and general *schema* for modeling information in the repository using generalization and aggregation in an object class diagram. This general schema has been derived from a conceptual modeling language originally defined in Telos (Mylopoulos, Borgida, Jarke, & Koubarakis, 1990). The general schema allows to define customized schema to model the information for different analysis purposes.

Due to their expressiveness and mathematical foundation, graphs are commonly used as the representation model for the software systems. All the above schemas or domain models define the types for system entities and their relationships which allow to represent the software system as a typed, attributed, directed graph. In this connection, the research on GXL (Graph eXchange Language) (Holt, Winter, & Schurr, 2000) is aimed at providing a standard exchange format for graph-based tools. The design of GXL is based on XML (Extensible Mark-up Language) (44,) which is a simple and flexible text format that was originally designed to meet the challenges of large-scale electronic

publishing. The domain model for the GXL is defined using the DTD (Document Type Definition) of XML.

TECHNIQUES FOR ARCHITECTURAL RECONSTRUCTION

In the following Sections, two broad categories of the software architecture reconstruction techniques are discussed. In the first category, namely *clustering*, a technique generates architectural components by gradual grouping the related system entities. The clustering category comprise of: *automatic/semiautomatic clustering* techniques that collect the related parts of a software system into cohesive components using a proximity metric (Wiggerts, 1997; Lakhotia, 1997; Tzerpos & Holt, 1998); *concept lattice* techniques that aggregate the groups of maximally related entities, arranged in the neighboring nodes of a concept lattice (Siff & Reps, 1999; Lindig & Snelting, 1997; Deursen & Kuipers, 1999); *composition and visualization* techniques that recover the containment-hierarchy of the system components using visualization and graph manipulation methods (Muller, Orgun, et al., 1993; Finnigan et al., 1997); and *data mining* techniques that discover the groups of entities that are related by association relation (Sartipi, Kontogiannis, & Mavaddat, 2000a; Miller & Gujarathi, 1999).

In the second category, namely *pattern-based*, a technique initiates by composing a high-level view of the system architecture (also known as the conceptual architecture or architectural pattern) using a modeling means, and then a search engine identifies the pattern in the software representation. The pattern-based techniques comprise of: *pattern*

matching techniques that model the high-level view of the system using a pattern modeling means and use approximate matching techniques to recover the pattern; *compliance checking* techniques that check the degree of conformance between a pattern and source-code; and *constraint checking* techniques that identify groups of entities that satisfy the constraints defined among them.

Clustering Techniques

The cluster analysis is defined as the process of classifying entities into subsets that have meaning in the context of a particular problem (Jain, 1988). The clustering techniques are designed to extract groups of related entities. The choice of a technique affects the detected clusters, which may or may not relate to the actual or intended structure of the system. These techniques provide potential and tractable means to identify cohesive system components.

Requirements for a Clustering Technique. In this Section, we discuss the requirements of clustering techniques for architectural reconstruction, as these have been presented in the related literature. These include: i) type of entities to be grouped; ii) similarity measure between two entities; and iii) clustering algorithm. Wiggerts (Wiggerts, 1997), Anquetil (Anquetil & Lethbridge, 1999), and Tzerpos (Tzerpos & Holt, 1998) have surveyed different aspects of clustering algorithms for software systems.

- **Entities to be clustered**. In clustering analysis, the granularity level of the selected source code entities depends on the purpose of the analysis. For example, function, datatype, and variable (lower-level of granularity) are usually used for clustering at the

module level, and file (higher-level of granularity) is used for clustering at the subsystem level. The application of domain model in extracting a suitable source model for the architectural reconstruction purpose was discussed earlier.

- **Similarity measure**. A *similarity* measure is defined so that two entities that are alike would possess a higher similarity value than two entities that are not alike. Different methods for similarity measure fall into two general categories. The first category is based on *relationships* between the entities (e.g., function call, or data use) where the similarity is measured as a function of the number of static occurrences of such relationships. The second category is based on *shared properties* (namely *features*) between two entities, where the similarity is measured based on the number of shared features. Patel (Patel, Chu, & Baxter, 1992) provides an interesting social relation analogy between "finding similar entities to an entity" and "finding the friends of a person in a party". Wiggerts provides a summary of different categories namely *association coefficients, correlation coefficients*, and *probabilistic measures* (Wiggerts, 1997). An evaluation of these similarity metrics can be found in (Davey & Burd, 2000). 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 (Everitt, 1993) such as *Jaccard* and *matching coefficient.*

- **Clustering algorithms**. Important clustering algorithms that apply to the field of software reverse engineering can be categorized as: i) *hierarchical algorithms*, where each entity is first placed in a separate cluster and then gradually the clusters are merged into larger and larger clusters until all entities belong to a single cluster; ii) *optimization*

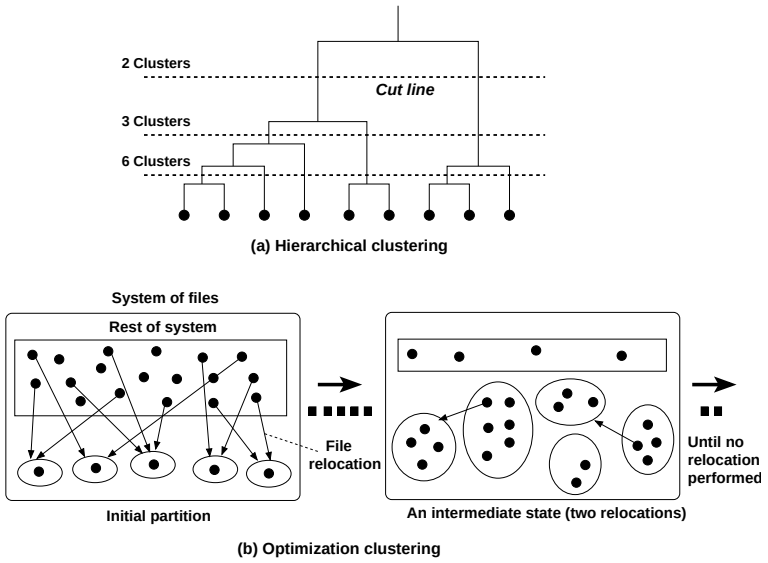

Figure 4. (a) Hierarchical clustering, where the cut-lines generate different number of clusters. (b) Optimization clustering (partitioning), where the entities in an initial partition are relocated among clusters based on some criteria until the partition is stable and no relocation is performed any more.

algorithms, where a partitioning of the whole system into clusters is considered and with iterative entity relocation among the clusters the partition is improved towards an optimal partition; and iii) *graph-theoretic algorithms*, where an entity relationship graph of the system is considered and the algorithm searches to find subgraphs with special properties such as maximal connected subgraphs or minimal spanning trees. A *supervised* clustering technique requires guides from the user in different stages to perform the clustering, whereas, an *un-supervised* clustering only relies on the similarity matrix consisting of the similarity of every pair of entities (Jain, 1988). Figure 4 illustrates examples of hierarchical clustering and optimization clustering (also called partitioning).

Automatic and Semi-Automatic Clustering Techniques. A technique in this group (Sartipi & Kontogiannis, 2001; Koschke, 1999; Anquetil & Lethbridge, 1999; Canfora,

Czeranski, & Koschke, 2000; Davey & Burd, 2000; Mancoridis, Mitchell, Rorres, Chen, & Gansner, 1998; Hutchens & Basili, 1985; Kunz & Black, 1995) uses a similarity metric (e.g., association coefficient, correlation coefficient, or probabilistic measures) which reflects a particular property among the system entities, and a clustering algorithm (e.g., agglomerative, optimization, graph-based, or construction) to partition the system into groups of related entities (Wiggerts, 1997). Lakhotia (Lakhotia, 1997) provides a unified framework that categorizes and compares the different software clustering techniques. In (Mancoridis et al., 1998) 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 (Tzerpos & Holt, 2000), 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 a specific number (e.g., 20 links) to/from a file are disregarded.

Concept Lattice Analysis. The *mathematical concept analysis* was first introduced by Birkhoff in 1940 (Birkhoff, 1940). 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 (Siff & Reps, 1999; Lindig & Snelting, 1997; Deursen & Kuipers, 1999). In such applications, a *formal concept* is a maximal collection of objects (i.e., system functions) sharing maximal common

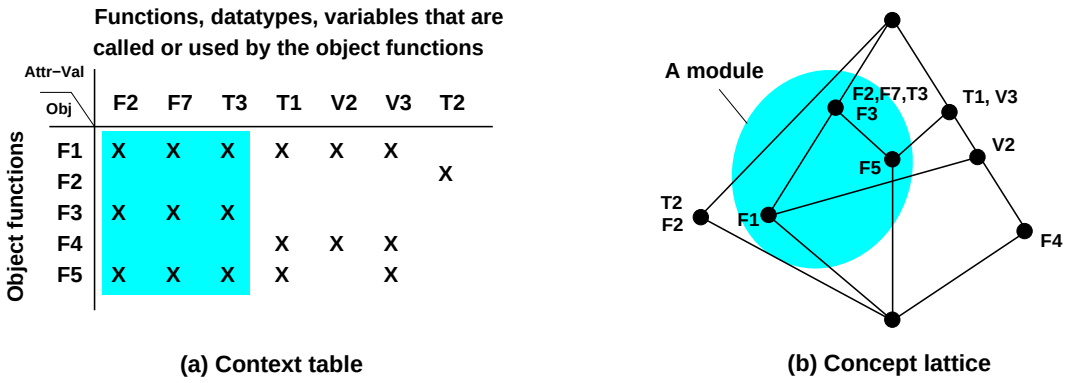

Figure 5. The application of concept lattice analysis in grouping the system entities into modules. The grey parts are two representation of the same group of entities, i.e., F1, F3, F5, F2, F7, T3.

attribute-values (i.e., called/used functions, datatypes, variables). A *concept lattice* can be composed to provide significant insight into the structure of the relations between objects and attribute-values such that each node of the lattice represents a concept.

The steps of using concept lattice for the modularization of a software system have been presented in (Siff & Reps, 1999) as follows. First, a matrix of functions and their attribute-values is built that is called a *context table*. 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 significant overlap. In this context, each partition corresponds to a potential decomposition of the system into modules. Figure 5 illustrates the application of concept lattice analysis in collecting a group of system entities that exist in neighboring concepts in a lattice. This group of entities constitutes a high cohesive module.

However, even in medium software systems (+50 KLOC) the concept lattice may become 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. Anquetil (Anquetil, 2000) addresses this problem and argues that concept lattice produces much more information, i.e., concepts, than it was given as input (i.e., a set of entities describing the software system), and hence he proposes a technique that only extracts the most interesting concepts. Godin (Godin & Mili, 1993) proposes a solution to the overwhelming number of concepts by pruning the concepts that do not introduce any attributes of their own, from the lattice of concepts. The pruned concepts are considered as non-important concepts. In the resulting lattice, each concept has at least one new attribute. Van Deursen (Deursen & Kuipers, 1999) proposes a technique to simplify the data set before extracting the concepts, e.g., by eliminating the programs with a high number of fan-in or fan-out. This approach uses the concept lattice to extract components in the term of classes of objects in the Cobol programs.

Visualization Techniques. The approaches in this group (Finnigan et al., 1997; Muller et al., 1993; Storey, Best, & Michaud, 2001) are based on tool usage, domain knowledge, and visualization means, to perform an iterative user-assisted clustering process. Such user-assisted techniques have been proven useful in handling large systems (Finnigan et al., 1997).

In the PBS approach (Finnigan et al., 1997), the user defines a containment structure for a hierarchy of subsystems which is derived from: developers, documentation,

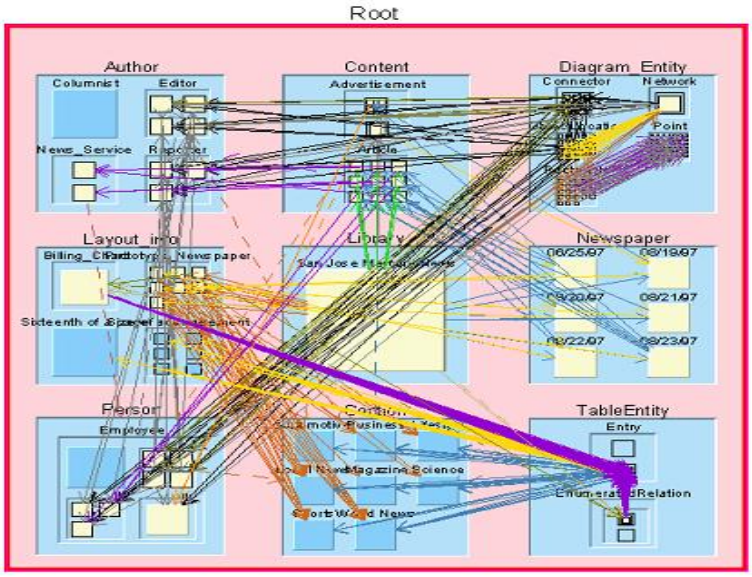

Figure 6. A structural view of a software system generated by the SHriMP software visualization tool.

directory structure, and naming conventions. The tool consequently reveals the relations between subsystems and represent the system architecture as "landscapes" in HTML pages for the user's inspection and manipulation. In the Rigi tool (Muller et al., 1993), the extracted facts in the form of RSF tuples are represented as an entity-relation graph of attributed boxes and relationship links. Interactive facilities for graph filtering and clustering operations to build and explore subsystem hierarchies are also provided.

SHriMP (Storey et al., 2001) is an information visualization and navigation system that employs zooming features to provide insight into the structure of the system under analysis. The fish-eye zooming feature allows the user to zoom on a particular piece of the software, while preserving the context of information. The tool uses search algorithms that allows the user to find and visualize the intended artifact in the system. The process

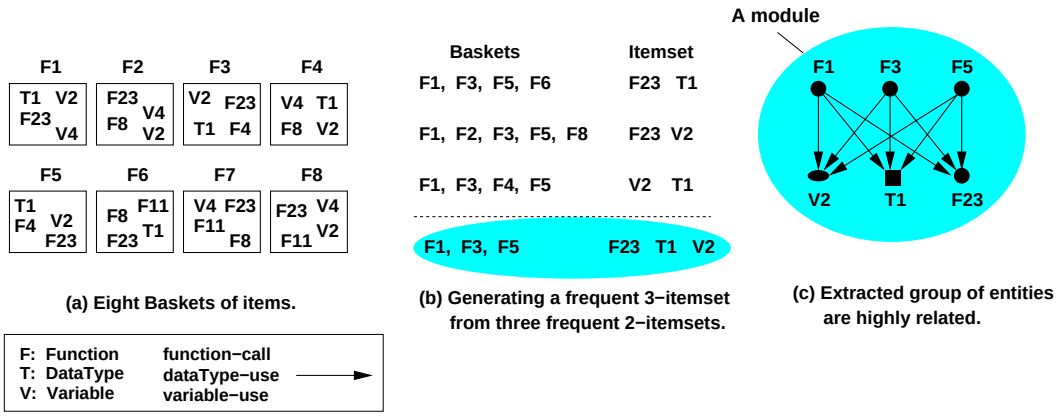

Figure 7. The application of data mining algorithms to discover group of highly related entities. (a) Representation of functions as baskets and called functions and used datatypes/variables as items in the baskets. (b) Representation of an iterative algorithm for generating frequent-itemsets. (c) A discovered group of highly related entities constitute a cohesive module.

is assisted by the user in order to construct high-level views of a system by grouping the elements in a graph. Figure 6 illustrates a view of the software system structure with links between different subsystems that has been generated by the SHriMP visualization tool.

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 (Fayyad, 1996). A substantial number of data mining approaches in the related literature are based on extensions of the *Apriori algorithm* by Agrawal (Agrawal & Srikant, 1994). These approaches are pertinent to the concept of *market baskets* (or *transactions*). 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 *association rules* (Agrawal & Srikant, 1994) express the frequency of pattern

occurrences such as 40% of baskets that contain the set of items {A,B} also contain the set of items {C,D}. The association rules can be extracted by the iterative algorithm Apriori 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 such frequent itemsets. The general idea is that if, for example, {A,B,C,D} and {A,B} are frequent itemsets, then the association rule $\{A, B\} \Rightarrow \{C, D\}$ can be extracted 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 association rule holds only if $r >$ *minimum confidence*.

The reverse engineering approaches using data mining are very few. Montes de Oca and Carver (Oca & Carver, 1998) 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. Miller and Gujarathi (Miller & Gujarathi, 1999) propose a knowledge discovery framework and work with association rules that essentially address the statistical information about relation between groups of entities in the database. Sartipi (Sartipi & Kontogiannis, 2001) uses a by-product of association rules, by considering frequent itemsets along with their container baskets. This information allows to encode the structural property of the groups of entities with maximum-level of interaction as a similarity measure between system entities. Figure 7 illustrates the steps for representing the software system entities as data mining baskets and their items, as well as a discovered group of highly related entities as the collection of the baskets and the corresponding itemset.

Pattern Based Techniques

The software architecture reconstruction techniques in this category are mostly designed as a top-down process where the high-level view of the system is built as the user's mental model of the system architecture. The pattern matching process then searches the source model which is either an abstract syntax tree, a repository of architectural elements, or a relational database to identify an exact or an approximate instance of the high-level view in the source model.

Modeling High-level System Specification. The high-level specification of a system, also called "architectural pattern" or "conceptual architecture" is an integral part of the pattern based architectural reconstruction techniques. The specification should represent an abstraction of the components and their interactions as well as a mechanism to constrain the type of the involved system entities and data/control dependencies. In a typical scenario for defining the high-level model of a system, the software engineer uses information from different sources such as: analytical data and statistical metrics on system properties provided by the toolkit; and knowledge about the application domain and the existing system documents.

In most cases, the high-level specification is defined as a query that specifies the expected architectural structure or behavior in the target system. An architectural query is defined using architectural notations, e.g., component, connector, module, and import/export. The notions such as *entities*, *connectivities*, and *scope* are used for structural view reconstruction. Whereas, the temporal operators are used to specify the expected

behavioral view of the system. The notions of event, event pattern, and event tracing, are intended for behavioral recovery. The query may use directives to lead the analysis algorithm to a specific architectural analysis task.

In these techniques, the high-level view of the system is modeled either as a collection of architectural styles (Harris, Reubenstein, & Yeh, 1995), a graph of architectural elements (Kazman & Burth, 1998), an ADL based query (Sartipi, 2003), an SQL based query (Kazman & Carriere, 1999), or an XML based query (Pinzger & Gall, 2002).

Pattern Matching Techniques. Sartipi (Sartipi, 2003) proposes a pattern-based architecture reconstruction approach that uses an architecture query language (AQL) to model the architectural pattern of the system as a constrained graph of components and connectors. The design of the AQL language has been inspired from the current architecture description languages (ADL). In this approach, a graph matching engine incrementally expands the architectural pattern (defined in AQL language) into a pattern graph and matches it with the system graph. The search algorithm uses a graph distance measure computed using costs of node/edge deletion/insertion to find a sub-optimal match as the recovered architecture.

Kazman and Burth (Kazman & Burth, 1998) 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 (Kazman,

Clements, Abowd, & Bass, 1997). 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 source model. The tool uses the constraint satisfaction paradigm (Woods & Yang, 1995) to restrict the search space. The hard/soft features of the elements allow to relax the exact matching in order to perform approximate matching. The approach provides statistics about the regularity of a system in terms of its coverage by a particular pattern.

Kazman and Carriere (Kazman & Carriere, 1999) propose *Dali* as a workbench that allows different light-weight tools and techniques to integrate for an architectural reconstruction 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. In this context, 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 reconstruction 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.

Harris et al. (Harris et al., 1995) identify architectural styles (about nine styles) in the source model. 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. (Fiutem, Merlo, Antoniol, & Tonella, 1996; Fiutem, Tonella, Antoniol, & Merlo, 1996) use "recognizers" and flow analysis techniques in architectural reconstruction.

Compliance Checking Techniques. In these techniques, 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, architectural style, or query). The tool then checks the degree of conformance between the proposed model and the source model. The following approaches are examples of compliance checking techniques.

Murphy and Notkin (Murphy, Notkin, & Sullivan, 1995) have proposed the software Reflexion model to assist the user in testing whether his/her mental model of the system conforms with 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.

Kontogiannis (Kontogiannis, DeMori, Bernstein, Galler, & Merlo, 1995) proposes a stochastic approach for structure compliance checking which is based on the notion of concept-to-code mapping. In this approach, a concept language models abstract properties of a desired code fragment. The pattern matching process is based on the *Markov* model and a 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.

Constraint Checking Techniques. Software architecture reconstruction has also been considered 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 (Woods & Yang, 1995) generalizes the problem of program understanding as an instance of the constraint satisfaction problem. Other variations of CSP may consider "soft" constraints that can be violated to a certain extent.

Sartipi et al. (Sartipi, Kontogiannis, & Mavaddat, 2000b) proposes an approach to software architecture reconstruction that uses an extension to the CSP problem known as *Valued Constraint Satisfaction Problem* framework (VCSP) (Schiex, Fargier, & Verfaillie, 1995), 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.

SCALABILITY OF THE RECONSTRUCTION PROCESS

The large size and the complexity of relations among the system entities are the sources of problem in dealing with large systems. The main idea is to decrease the domain space in searching for architectural information without losing relevant information. In this connection, in (Sartipi, 2003) several heuristics have been proposed that are briefly discussed below.

Incremental Reconstruction

An effective heuristic in decreasing the time and space complexity of the search process is to divide the whole search space into sub-spaces and then perform incremental reconstruction process where the architectural components are recovered one at a time. The heuristic consists of two steps. In the first step, the search space is divided into many sub-spaces according to the particular property. For example: a sub-space can be a group of associated entities if the objective is to recovered components that are highly associated; or a sub-space can be a group of functions that perform data read and write on specific files in order to recover "filter" components. Then at each iteration of the incremental reconstruction process, one sub-spaces is selected according to its eligibility

which is determined by a ranking mechanism to choose the best candidate sub-space.

Sub-Optimal Reconstruction Process

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 make a trade-off between the quality of the recovered components and the search complexity. In this context, some researchers use non-complete and non-optimal but fast search techniques such as *hill climbing* (Mancoridis et al., 1998). In (Sartipi & Kontogiannis, 2001) a heuristic version of the optimal search algorithm A^* is proposed that significantly reduces the time/space requirement of the A^* search with the cost of having a sub-optimal reconstruction process. Also, approximate matching is another technique that uses a cost function and recover the architectures that are close to an intended architecture within a boundary of a particular threshold (Kazman & Burth, 1998).

Hierarchical Reconstruction Process

The architectural reconstruction process of large systems with several hundreds thousands of lines of code such as Linux operating system (Bowman, Holt, & Brewster, 1999) is usually limited to activities such as consulting with the existing system documents, relying on the exactness of the naming conventions (if exist) for files and directories, and the directory structure. In such cases, the lowest granularity of the system entities is file and directory. However, a detailed analysis is not directly feasible where the lowest granularity are function, user-defined aggregate types, and global variables.

The hierarchical reconstruction of large systems usually consists of three levels, as: i) decomposing the system into a few large subsystems; ii) decomposing each large subsystem into a number of smaller subsystems of files and directories; and iii) decomposing each small subsystem into a number of modules of functions, aggregate/array types, and global variables.

USER INVOLVEMENT IN THE RECONSTRUCTION PROCESS

In the current approaches to architectural reconstruction the role of user is increasingly important in order to incorporate design-specific criteria in the process of structure reconstruction of a software system. Such design-specific criteria can not be fully formulated in order to be automatically investigated by the analysis program. 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 reconstruction process (Finnigan et al., 1997; Chin & Quilici, 1996). In this context, the new terms such as *librarian* and *patron* (Finnigan et al., 1997) refer to the system information accumulation for human usage. In such cases, the analysis tool, as the user assistance, must process the raw information that represents the software system so that the user, as the high-level decision maker, can interpret and assess the processed information, in order to get insight into the system and also to make decision for the next step of the reconstruction. Examples of such information include:

- *Statistical metrics.* Association relation among the system files; fan-in and fan-out; and architectural design views (Sartipi, 2001b).

- *Visualization means.* Simplifying the graph views (Sartipi & Kontogiannis, 2002); Source code browsing mechanism through HTML pages (Finnigan et al., 1997; Storey et al., 2001; Sartipi & Kontogiannis, 2002).

- *Pattern generation.* Analyzing the system representation database in order to identify the locus of interactions among system entities; this would allow the user to select the cores of functionality of the system via a ranked list (Sartipi, 2003).

ARCHITECTURAL EVALUATION TECHNIQUES

Evaluating the result of the reconstruction process is a debating issue with no generally accepted evaluation criteria. Different characteristics of the recovered architecture may be tested, including, modularity quality of the architecture through coupling and cohesion metrics, and the non-functional qualities. One important characteristic of the recovered architecture to evaluate is the accuracy of the reconstruction technique that can be assessed using the information retrieval metrics Precision and Recall (Grossman & Frieder, 1998). These metrics assess the compatibility of the recovered architecture with the documented system architecture. In this evaluation, the software system must possess an updated architectural document as the reference architecture. Figure 8 illustrates the accuracy assessment of the recovered subsystems for the Clips system. Clips is an expert system builder with 40 Kilo lines of code that is supported by a complete architectural manual (Section, 1989). The evaluation steps are discussed as follows. First,

No. of files in recovered subsystems	Subsystems specified in CLIPS reference architecture	No. of files in reference subsystems	No. of shared files among recovered & reference subsystems	Precision	Recall
S1: 11	– Defrule structures – Inference engine	13	9	82%	70%
S2: 10	– Rule manipulation	6	5	50%	83%
S3: 4	– Object	3	3	75%	100%
S4: 4	– Expression evaluation	4	3	75%	75%
S5: 10	– System function – User interface	7	4	40%	57%

Figure 8. Architectural reconstruction and evaluation of the Clips system.

the subsystems of the reference architecture must be identified. Second, the recovered subsystems must be matched against the subsystems from the reference architecture. It is common that the reference and recovered subsystems overlap to some extent, or one or more recovered subsystems may partially fit in one reference subsystem (or vice versa). In such cases we implicitly merge the subsystems into one to allow almost one-to-one comparison between subsystems. Third, the Precision and Recall metrics are computed for each recovered subsystem. Precision is defined as the percentage of the number of the "shared files" in the corresponding recovered and reference subsystem, to the number of "recovered files" for that subsystem. Whereas, Recall is defined as the percentage of the number of "shared files" to the number of "reference" files for that subsystem. Figure 8 presents the evaluation of the reconstruction of the Clips system. In overall, such values of Precision and Recall indicate a promising reconstruction result.

Validation of the Reconstruction Approach

In this section, three evaluation techniques are discussed which are based on the level of conformance between the elements of the recovered components (i.e., subsystems

of files or modules of functions, types, variables), namely *candidate* components, and the elements of the *reference* component. The evaluation computation in these techniques are extensions of the Precision and Recall measures that were discussed above.

Lakhotia (Lakhotia & Gravley, 1995) evaluates the level of agreements between individual pairs of components in candidate and reference components in a hierarchical subsystem clustering. The metric, called *congruence measure*, uses both: difference between component similarities, and overlap between component elements among the candidate and reference component.

Koschke (Koschke & Eisenbarth, 2000) proposes an evaluation metric for non-hierarchical clustering, that is only based on the degree of overlap between the corresponding components in candidate and reference components. The technique is an extension to the Lakhotia's method by measuring the overall *accuracy* and *recall* for the whole clustering result, as opposed to separately comparing different pairs in Lakhotia's method. The reference components are subjectively determined based on a set of guidelines to the software engineer. In (Koschke & Eisenbarth, 2000), different cases such as 1-to-n or n-to-1 relation between the matching of candidate and reference components are also considered. Koschke defines the overall *accuracy* of the reconstruction process as:

$$accuracy(M) = \frac{\sum_{(C,R)\in M} \frac{elements(C) \ \cap \ elements(R)}{elements(C) \ \cup \ elements(R)}}{|M|}$$

where, C and R represent a pair of matching candidate and reference components and M is the set of matching pairs (C, R). Considering the reconstruction example of Figure 8,

the accuracy of the reconstruction is computed as follows:

$$accuracy(M) = \frac{\frac{9}{15} + \frac{5}{11} + \frac{3}{4} + \frac{3}{5} + \frac{4}{13}}{5} = 0.586$$

Mitchell (Mitchell & Mancoridis, 2001) proposes an automated method and a tool to extract the reference components in the absence of a benchmark decomposition that is used by the Koschke's method. A set of clustering techniques are used to produce different clustering results which are separately stored in a database, as pairs of "a component, an element of component". A further analysis of the tuples generates the reference components.

FUTURE TRENDS

The variety, size, and complexity of the modern software systems are continuously increasing to meet the new demands in different industrial applications such as, telecommunications, automotive, banking, insurance, medical, and air-traffic control. As a result, the current maintenance techniques (in particular reverse engineering and architecture reconstruction techniques) must be more sophisticated in order to tackle the new challenges that the software maintainers will face. In this respect, several research areas as the potential future trends in the field of software architecture reconstruction techniques are presented below.

- *Dynamic run-time analysis.* Research on dynamic aspects of a software system is more challenging than research on structural analysis. This may be due to the difficulty of

defining proper task scenarios that generate execution traces, and the difficulty of finding actual patterns in the generated executions traces. Therefore, more research on how to extract and how to use dynamic information for behavioral recovery is required.

- *Distributed and heterogeneous systems.* With the rapid growth of distributed and network-centric systems, architecture reconstruction techniques and tools should tackle the non-monolithic and heterogeneous systems that are operating in platforms with different hardware and software requirements, programming languages, and architectural patterns. In this respect, more research is needed in analyzing distributed multi-language multi-platform systems.

- *Dealing with large scale systems.* Currently, the architecture reconstruction of large systems mostly rely on non-formal facts such as naming convention and directory structure. More research on tractable heuristics for hierarchical architecture reconstruction would allow to incorporate structured information generated by extraction techniques at different levels, that is at function-level, file-level, and directory level.

- *Consistency and traceability among views.* Most reconstruction techniques focus on only one view of the system to recover. New techniques based on multiple view reconstruction or multiple abstraction-level reconstruction techniques would provide much deeper understanding about the system. In such techniques, maintaining the consistency and traceability among the views or abstraction levels would be of significant importance.

- *Reconstruction techniques for the development process.* This research aims to assist the project managements to ensure that the design objectives and requirements are met during the implementation or the evolution of a system.

- *Information exchange.* Because of the expressiveness and mathematical foundation of graphs, most of the reconstruction approaches use a customized attributed graph representation for the software system. The research activities for standardization of the software representation have been focused on using XML (extensible markup language) to provide a customizable typed attributed graph representation for different software artifacts. This standard graph can be exchanged among different tools for the purpose of information extraction, architectural reconstruction, and analysis of the software systems. This new trend has already attracted lots of attention among the researchers and the project GXL (graph exchange language) has been launched to achieve this research goal (Holt et al., 2000).

CONCLUSION

The software technology is evolving and new methodologies, techniques, and tools are emerging on the quest for better design, implementation, and maintenance of large and mission critical software. However, when the software system is operational its functionality constantly evolves and, if not maintained properly, in most cases its current structure drifts form its documented and intended structure. In such cases, any well designed software system becomes a legacy system which costs a lot for the organization to operate reliably. As an alternative to the costly replacement of such systems, the organizations may choose to re-engineer or re-structure the systems. In these maintenance activities, reverse engineering or architecture reconstruction is performed first to allow the engineers understand the structure or behavior of the software system. The scope of re-

search in software architecture reconstruction techniques and tools spans several research areas, including: compilers, profilers, programming languages, clustering, concept lattice analysis, pattern recognition, data mining, graph theoretic, constraint programming, and graphical user interfaces.

The notion of software views allows to achieve separation of concern in the reconstruction process by classifying the set of features that are relevant to the structure, behavior, or environment views. In this Chapter, we attempted to cover the techniques for reconstruction of the structure view of a software system. The approaches to architectural reconstruction are expected to address the following issues: i) the view(s) of the system to be recovered, where the structure view is the most common view to consider; ii) the software representation model, that specifies the intended granularity of the system entities and their relations defined by a domain model or schema; iii) the adopted high-level model of the software system, which is particularly important in the pattern-based reconstruction approaches; iv) the employed architecture reconstruction technique, which is generally categorized as either a clustering-based technique which tends to be automated, or a pattern-based technique which extensively relies on user for pattern definition; v) the tractability of the reconstruction process, that is particularly important in dealing with large systems where the heuristics may be used to trade the tractability of the process against the quality of the result; and vi) the method of evaluating the result of reconstruction which is significantly important for the research community to assess the variety of existing or emerging approaches.

Finally, in the discussions of this Chapter it was attempted to expose the reader to

a systematic approach of the important issues, alternative solutions, and future research, in the field of software architecture analysis and reconstruction. We believe that this Chapter will provide enough insight into this exciting area that can be used as a starting point by the interested reader to perform further research in this area.

References

(Rigi, URL = http://www.rigi.csc.uvic.ca/index.html)

(XML, Extensible Markup Language,
URL = http://www.w3.org/XML/)

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th international conference on very large databases* (p. 487-499).

Anquetil, N. (2000). A comparison of graphs of concept for reverse engineering. In *Proceedings of the international workshop on program comprehension (iwpc'00)* (p. 231-240).

Anquetil, N., & Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization. In *Proceedings of the sixth working conference on reverse engineering* (p. 235-255).

Birkhoff, G. (1940). *Lattice theory* (1st ed.). American Mathematical Society.

Bojic, D., & Velasevic, D. (2000). A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *Proceedings of the csmr'00* (p. 23-31).

Bowman, I. T., Holt, R., & Brewster, N. (1999). Linux as a case study: its extracted software architecture. In *Proceedings of the icse'99* (p. 555-563).

Canfora, G., Czeranski, J., & Koschke, R. (2000). Revisiting the delta ic approach to component recovery. In *Proceedings of the wcre'00* (p. 140-149).

Chen, Y.-F., Nishimoto, M. Y., & Ramamoorthy, C. V. (1990). The c information abstraction system. *IEEE Transaction on Software Engineering*, *16*(3), 325-334.

Chin, D. N., & Quilici, A. (1996). Decode: A co-operative program understanding environment. *Software Maintenance: Research and Practice*, *8*, 3-33.

Davey, J., & Burd, E. (2000). Evaluating the suitability of data clustering for software remodularisation. In *Proceedings of the seventh working conference on reverse engineering* (p. 268-276).

Deursen, A. van, & Kuipers, T. (1999). Identifying objects using cluster and concept analysis. In *Proceedings of the icse 1999* (p. 246-255).

Eisenbarth, T., Koschke, R., & Simon, D. (2001). Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the iwpc'01* (p. 300-309).

El-Ramly, M., Stroulia, E., & Sorenson, P. (2002). Mining system-user interaction traces for use case models. In *Proceedings of iwpc'02* (p. 21-29).

Everitt, B. S. (1993). *Cluster analysis.* John Wiley.

Fayyad, U. M. (1996). *Advances in knowledge discovery and data mining.* Menlo Park, Calif.: MIT Press.

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

Fiutem, R., Merlo, E., Antoniol, G., & Tonella, P. (1996). Understanding the architecture of software systems. In *Proceedings of the 4th workshop on program comprehension* (p. 187-196).

Fiutem, R., Tonella, P., Antoniol, G., & Merlo, E. (1996). A cliche-based environment to support

architectural reverse engineering. In *Ieee international conference on software maintenance (icsm)* (p. 319-328).

Godin, R., & Mili, H. (1993). Building and maintaining analysis-level class hierarchies using galois lattices. *ACM SIGPLAN Notices, ACM Press, New York, NY, USA*, *28*(10), 94 - 410.

Grossman, D. A., & Frieder, O. (1998). *Information retrieval: algorithms and heuristics.* Kluwer Academic Publishers.

Harris, D. R., Reubenstein, H. B., & Yeh, A. S. (1995). Reverse engineering to the architectural level. In *Proceedings of the 17th icse* (p. 186-195).

Holt, R., Winter, A., & Schurr, A. (2000). Gxl: Toward a standard exchange format. In *Proceedings of the working conference on reverse engineering* (p. 162-171).

Hutchens, D. H., & Basili, V. R. (1985). System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, *SE-11*(8), 749-757.

Jain, A. K. (1988). *Algorithms for clustering data.* Englewood Cliffs, N.J.: Prentice Hall.

Kazman, R., & Burth, M. (1998). Assessing architectural complexity. In *Proceedings of the csmr* (p. 104-112).

Kazman, R., & Carriere, S. J. (1999). Playing detective: Reconstruction software architecture from available evidence. *Journal of Automated Software Engineering*, *6*(2), 107-138.

Kazman, R., Clements, P., Abowd, G., & Bass, L. (1997). Classifying architectural elements as a foundation for mechanism matching. In *Proceedings of the compsac* (p. 14-17).

Kontogiannis, K., DeMori, R., Bernstein, M., Galler, M., & Merlo, E. (1995). Pattern matching for design concept localization. In *Proceedings of the working conference on reverse engineering (wcre'95)* (p. 96-103).

Koschke, R. (1999). An incremental semi-automatic method for component recovery. In *Proceedings of the sixth working conference on reverse engineering* (p. 256-267).

Koschke, R., & Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. In *Proceedings of the iwpc* (p. 201-210).

Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE Software*, *12*(6), 42-50.

Kunz, T., & Black, J. P. (1995). Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, *21*(6), 515-527.

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

Lakhotia, A., & Gravley, J. (1995). Toward experimental evaluation of sybsystem classification recovery techniques. In *Proceedings of the wcre* (p. 262-269).

Lindig, C., & Snelting, G. (1997). Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th international conference on software engineering* (p. 349-359).

Mancoridis, S., Mitchell, B., Rorres, C., Chen, Y., & Gansner, E. (1998). Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the iwpc* (p. 45-53).

Miller, R. J., & Gujarathi, A. (1999). Mining for program structure. *International Journal on Software Engineering and Knowledge Engineering*, *9*(5), 499-517.

Mitchell, B. S., & Mancoridis, S. (2001). Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of the wcre* (p. 93-102).

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

Murphy, G. C., Notkin, D., & Sullivan, K. (1995). Software reflexion model: Bridging the gap between source and higher-level models. In *In proceedings of the 3rd acm sigsoft sfse* (p. 18-28).

Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1990). Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, *8*(4), 325-362.

Oca, C. M. de, & Carver, D. L. (1998). A visual representation model for software subsystem decomposition. In *Proceedings of the working conference on reverse engineering* (p. 231-240).

Patel, S., Chu, W., & Baxter, R. (1992). A measure for composite module cohesion. In *International conference on software engineering* (p. 38-48).

Pinzger, M., & Gall, H. (2002). Pattern-supported architecture recovery. In *Proceedings of the ieee iwpc'02* (p. 53-61).

Poulin, J. S. (1996). Evolution of a software architecture for management information systems. In *Proceedings of the second international software architecture workshop (isaw-2)* (p. 134-137).

Sartipi, K. (2001a). Alborz: A query-based tool for software architecture recovery. In *Proceedings of the ieee international workshop on program comprehension (iwpc'01)* (p. 115-116). Toronto, Canada.

Sartipi, K. (2001b). A software evaluation model using component association views. In *Proceedings of the ieee international workshop on program comprehension (iwpc'01)* (p. 259-268). Toronto, Canada.

Sartipi, K. (2003). *Software architecture recovery based on pattern matching.* Unpublished doctoral dissertation, School of Computer Science, University of Waterloo, Waterloo, ON, Canada.

Sartipi, K., & Kontogiannis, K. (2001). Component clustering based on maximal association. In *Proceedings of the ieee working conference on reverse engineering (wcre'01)* (p. 103-114). Stuttgart, Germany.

Sartipi, K., & Kontogiannis, K. (2002). A user-assisted approach to component clustering. *Accepted for the Journal of Software Maintenance: Research and Practice (JSM).*

Sartipi, K., & Kontogiannis, K. (2003). On modeling software architecture recovery as graph matching. In *"to appear" in the proceedings of icsm'03* (pp. –).

Sartipi, K., Kontogiannis, K., & Mavaddat, F. (2000a). Architectural design recovery using data mining techniques. In *Proceedings of ieee csmr 2000* (p. 129-139). Zurich, Switzerland.

Sartipi, K., Kontogiannis, K., & Mavaddat, F. (2000b). A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of the ieee iwpc* (p. 37-47). Limerick, Ireland.

Schiex, T., Fargier, H., & Verfaillie, G. (1995). Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the ijcai-95* (p. 631-637).

Section, A. I. (1989, May). *Clips architectural manual version 4.3.*

Siff, M., & Reps, T. (1999). Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, *25*(6), 749-768.

Software development kit: Refine/c user's guide for version 1.2. (1998, April). 700 E. El Camino Real, Mountain View, CA 94040, USA.

Soni, D., Nord, R. L., & Hofmeister, C. (1995). Software architecture in industrial applications. In *Proceedings of the 17th international conference on software engineering* (p. 196-207).

Storey, M.-A., Best, C., & Michaud, J. (2001). Shrimp views, an interactive environment for exploring java programs. In *Proceedings of iwpc'01* (p. 111-112).

Tzerpos, V., & Holt, R. C. (1998). Software botryology: Automatic clustering of software systems. In *Proceedings of the international workshop on large-scale software composition* (p. 811-818).

Tzerpos, V., & Holt, R. C. (2000). Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the seventh working conference on reverse engineering* (p. 258-267).

Wallmuller, E. (1994). *Software quality assurance: A practical approach.* New York: Prentice Hall.

Wiggerts, T. A. (1997). Using clustering algorithms in legacy systems modularization. In *Proceedings of the fourth working conference on reverse engineering* (p. 33-43).

Woods, S. G., & Yang, Q. (1995). Program understanding as constraint satisfaction. In *Proceedings of second working conference on reverse engineering* (p. 314-323).

Zachman, J. A. (1987). A framework for information systems architecture. *IBM Systems Journal*, *26*(3), 276-292.

Zaremski, A. M., & Wing, J. M. (1995). Specification matching of software components. *Proceedings of SIGSOFT 95 Software Engineering Notes*, *20*(4), 6-17.