A Software Evaluation Model Using Component Association Views *

Kamran Sartipi

Department of Computer Science,
University of Waterloo,
Waterloo, ON. N2L 3G1, Canada
ksartipi@math.uwaterloo.ca

Abstract

In this paper, we introduce a view-based architectural design evaluation model that allows to quantitatively evaluate and categorize the design of a software system. The model is based on the notion of component association which is a generalization of coupling and cohesion metrics. The component association is defined as a measure of the overall dependency among high-level system components such as files, modules, or subsystems with regard to a collection of criteria. The associations are discovered by applying data mining techniques on a database of data and control flow dependencies extracted from the software system. The proposed association-views and modularity metrics allow the user to evaluate the design quality of a software system.

1 Introduction

As an integral part of a reverse engineering task, the user investigates the system domain and documents to obtain insight into the system under analysis. To assist this task, various CASE tools provide meaningful and well presented metrics about the software system. We present a measuring technique for both inter- and intra-component interactions that has been inspired from the coupling and cohesion metrics. In this context a *component* is defined as a named group of system entities.

In a software system consisting of modules, *coupling* is a measure of the "relative interdependence among the modules" [15], and is measured based on the complexity of the interface between the modules [11]. The *cohesion* is a measure of the "relative functional strength of a module" [15] and is usually measured by techniques based on program slicing [6].

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

We obtain the coupling and cohesion properties among system components by measuring the inter-/intra-component associations, where the association is a property among highly related groups of entities in term of the maximum number of shared entities in the groups.

The proposed technique allows us to measure the modularity of the software system, as an indication of the quality of the system design and its decomposition into subsystems. In this context, we generate three almost orthogonal association views of a system denoted as *control passing*, *data exchange*, and *data sharing* views, each illustrating a projection of a graph of components and association links based on a particular data-/control-dependency property. Using these views, we consider three *design properties* for a system based on *sharing*, *passing*, or *encapsulating* the state of the system, regardless of the system's adopted design methodology and architectural style.

In this approach, the software system is modeled as an *attributed relational graph* with the system entities as nodes and data-/control-dependencies as edges. The application of data mining techniques on the system graph allows to decompose the graph into domains of entities based on the association property and then populate a database of these domains. An analysis of the domains generates a *component graph* where the edges represent the association strengths among the components, to be used for quality analysis.

2 Motivation

In the software engineering literature, coupling and cohesion properties have been defined using ordinal scales, namely: *data*, *stamp*, *control*, *external*, and *common* as coupling categories; and *coincidental*, *logical*, *temporal*, *procedural*, *communicational*, *sequential*, and *functional* as cohesion categories [15]. Both properties are used for decomposition of a system into modules of system entities. There are less controversial proposals for coupling measurement

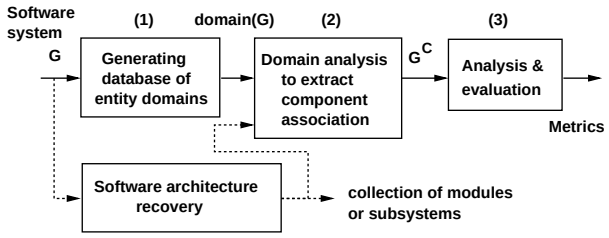


Figure 1. Three phases of the software evaluation model.

than cohesion measurement. The coupling proposals follow different paths but in the same spirit of the original definition as "relative interdependence among the modules" [11, 9, 14].

On the other hand, measuring the cohesion of a module is still challenged by different proposals [13]. Determining a module's cohesion based on the original definition usually requires a program slicing technique [6] which is computationally expensive. Therefore, a number of authors have adopted a different approach and view the cohesion as "*coherence*" [13] or "*intra-connectivity* [11] which is in fact a form of *external* property of a function as opposed to internal properties extracted by the slicing methods. The proposals in this group consider a number of shared features for each function and determine the cohesion as the degree of sharing different sets of features such as: global variables, function calls, or data types [10, 8]. In this form a cohesive module is produced by clustering the functions based on high *similarity* degree among them with respect to a set of shared features [9].

Considering a system of large components at the architectural level, measuring the coupling and cohesion metrics need to be generalized to capture the complexity of the inter- and intra-component interaction. The component interfaces usually cover different types of couplings which requires a form of averaging to present a single value to demonstrate the coupling among components. The same argument is valid for cohesion measurement.

Based on the above observations, we propose a single measuring technique to evaluate the coupling between two components or the cohesion of a component which is derived from important data and control flow dependencies between or within the components. The metric measures the association or relevance of the groups of entities in the components in term of the number of shared entities in a highly related group of entities. We call this metric *component association* and use it as a similarity measure between components in a system. In this paper, we use this similarity metric to analyze the quality of a system's design. In a future work, we will use this similarity metric in a clustering technique. In this model, the degree of association (or *relevance*) between two groups of functions is approximated as the overall coupling between two groups. Also, the degree of association of a group of function with themselves is approximated as the overall cohesion in that group.

3 Software evaluation model

The proposed approach to software evaluation based on component association consists of three phases (Figure 1).

In the first phase, the software system is parsed and presented as an attributed relational graph G_t [7], with nodes as source code entities (i.e., *file, function, type*, and *variable*), and edges as data and control flow dependencies (i.e., *call, define, set, update*, and *declare*). The low-level relations between entities are aggregated into more abstract relations (i.e., *call* and *use*) between the entities which are then presented as the source model graph G (Figure 2). Using data mining techniques, the entities in graph G are grouped based on the association property, and the result is represented as a collection of domains, denoted as $domain(G)$, where each domain corresponds to a system entity.

In the second phase, a domain analysis is performed on $domain(G)$, the association degrees among the system components are measured, and a *component graph* G^C or an *association view* is generated. In G^C a node is a system component and an edge is the association value between two components. In this form, a component refers to a group of the system entities in the form of a file (to evaluate a system design), or module and subsystem (to evaluate an architectural recovery task).

In the third phase, the component graph G^C is analyzed to provide metrics for assessing the quality of the software system or the result of a recovery task, as well as categorizing the system design.

3.1 Graph based system representation

In this section, we briefly introduce the underlying concepts of *Attributed Relational Graph* (ARG) that we use for representing a software system based on the notation used in [7].

An ARG is defined as a six-tuple $G = (N, R, A, E, \mu, \epsilon)$, where $N = \{n_1, n_2, ..., n_n\}$ is the set of attributed vertices (*nodes*), $R = \{r_1, r_2, ..., r_m\}$ is the set of directed attributed edges such that $R \subseteq N \times N$, A is an alphabet for node attributes, E is an alphabet for edge attributes, μ and ϵ are node and edge labeling functions for returning node and edge attributes, respectively. In a software system, typical node and edge attributes include:

- *label:* a string denoting a unique name for each entity in the software system, (i.e., a full path name). The edges do not have labels.

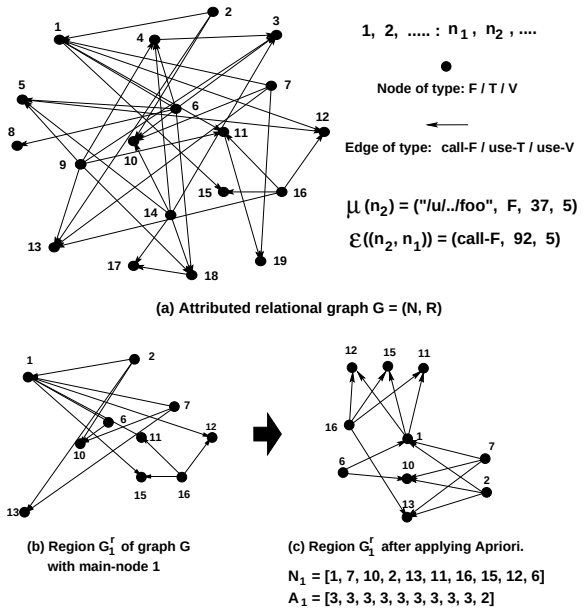


Figure 2. Application of data mining on the source model graph G.

- *type:* an identifier that classifies the nodes of a graph into different categories for nodes (e.g., L, F, T, V for *file, function, aggregate-type, global-variable*[1]), and also classifies the edges of a graph (e.g., *call-F, use-T*, and *use-V*).
- *location:* two integers for *file number* and *line number in file* for nodes and edges.

The *allowed edges* are defined using a triple (*node-type, edge-type, node-type*), e.g., (*F, use-T, T*) meaning "*function uses type*". Examples of node and edge labeling functions μ and ϵ in a software system include:
$\mu(n_{10}) = (\text{"/u/.../analysis.c/rule\_analysis"}, F, 79, 5)$ indicating that node 10 with the label *"/u/.../analysis.c/rule_analysis"* is of type F (function) and is defined in line 79 of the source file 5; and $\epsilon(r_{28}) = \epsilon((n_7, n_{10})) = (\textit{call-F}, 65, 13)$ with similar interpretation.

Figure 2 represents the ARG of a software system with 19 nodes (entities) demonstrating the complexity of a small system graph.

4 Association

Association in a graph is a property A among two or more source nodes that share one or more sink nodes (through graph edges). In this sense, the group of source and sink nodes are denoted as *associated group*. The *association degree* between the nodes of an associated group is the number of sink nodes, and the *association support* is the number of source nodes in that group. In Figure 2(c), the group of nodes 1, 7, 10, 2, 13 are associated with association degree 3 (i.e., 3 sink nodes) and association support 2 (i.e., 2 source nodes); and the group of nodes 6, 1, 7, 10, 2 are associated with degree and support 2 and 3, respectively.

Revealing all the associated groups in a large graph is a computationally expensive task. We use the Apriori algorithm [5] originally presented in the data mining domain to extract the groups with maximum association based on the notion of *frequent itemsets*. A more detailed discussion on the application of the data mining on reverse engineering can be found in [17, 12].

4.1 Graph region and domain

A *region* $G_j^r = (N_j^r, R_j^r)$ of a graph $G = (N, R)$ is a subgraph of G (i.e., $N_j^r \subseteq N$ and $R_j^r \subseteq R$) corresponding to a node $n_j \in N_j^r$. In a region G_j^r each node $n_i \neq n_j$ satisfies the association property A with respect to node n_j. We call n_j the *main-node* of the region G_j^r.

Figure 2(b) represents region G_1^r of the source model graph G that satisfies the association property, i.e., each node of G_1^r is a member of an associated group with respect to node 1. However, it is not clear what is the highest association degree of each node with regard to node 1, since each node can be a member of several associated groups having a different association degree in each group. The Apriori algorithm extracts all the associated groups in a region which allow us to determine the maximum association degree of each node with respect to the region's main-node. Figures 2(c) presents the application of the Apriori algorithm on the region in part (b). The nodes of a region are ranked according to the highest association degree with the main-node (Figures 2(c)).

Entity domain
The *domain* of a node n_j in graph $G = (N, R)$, denoted as D_j, is defined as: *"the collection of the graph nodes that are associated with node n_j along with their highest association degrees with respect to n_j"*. Formally,

$$D_j = \{(n_j, n_d, a) \mid n_j, n_d \in N_j^r \ \wedge \ n_j \textit{ is main-node of } G_j^r \ \wedge \ a = A_j(n_d)\}$$

Where, the function $A_j(n_d)$ returns the maximum association degree[2] of the node n_d in the region G_j^r.

[1] In the rest of paper we refer to *aggregate-type* and *global-variable* as *type* and *var*, respectively.

[2] In the rest of paper "*association degree*" of an entity refers to its "*maximum association degree*".

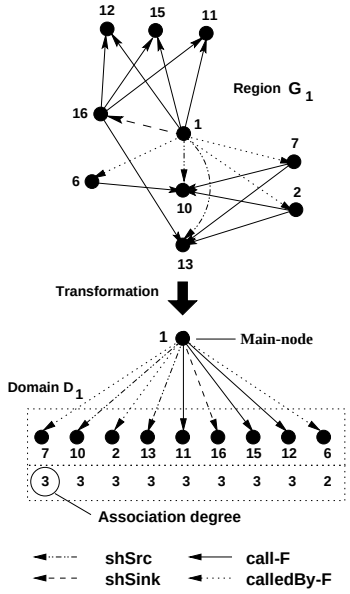


Figure 3. The transformation of region G_1 into domain D_1.

Figure 3, illustrates the transformation of the region G_1 into domain D_1. The transformation for each region is as follows:

- Two new edge types *shSink* and *shSrc* (denoted as *sharing sink node* and *sharing source node*, respectively), are added to each region.
- Each edge ending to the main-node is replaced by an edge starting from the main-node with inverse relation.
- A tree with main-node as the root and other region nodes as the leaves is built, where, the leaves are the domain of the root node with the relations indicated as the tree edges.

The *domain space* of the graph $G = (N, R)$, denoted as *domain(G)* is a database of all node domains D_j, which is defined as:

$$domain(G) = \bigcup_{j=1}^{|N|} D_j$$

4.2 Component

A system component is a named grouping of the system entities, such as function, type, and variable, with the relation *contains* or *defines* to those entities. Each system entity is contained in only one component. Each user defined entity in an *include file* (i.e., global variable or aggregate type) is contained in one component, based on the frequency of usage by the functions in that component. Library entities, defined using *include files*, can be viewed as an external component defined by the environment. A component can be a *file* or *module* of entities, or a *subsystem* of files where the files are replaced by their contained entities. Please note that the group of entities in a component is independent of a group of entities in an *entity domain.*

Assume that the source model graph $G = (N, R)$ is a composition of p components. We define a system *component* C_i ($i \leq p$) as a graph $G^{c_i} = (N^{c_i}, R^{c_i})$ which is a subgraph of G. The *domain of a component* C_i, denoted as D^{c_i}, is *a collection of the system entities that exist in the domain of each entity* n_j, *where the entity* n_j *is contained in component* C_i, Formally:

$$D^{c_i} = \{(n_j, n_d, a) \in domain(G) \mid n_j \in N^{c_i} \ \wedge \ n_d \in N\}$$

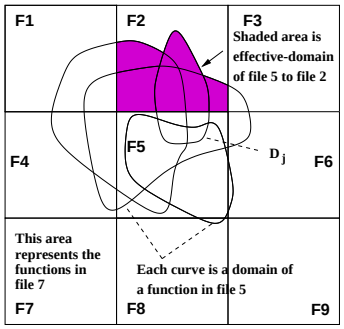
To simplify the concepts, we continue our discussion using file as a component, where each file contains (defines) a number of functions. In Figure 4, the domain of each entity (function) in file $F5$ is shown as the area in a closed curve. The domain of file $F5$ (i.e., D^{c_5}) is represented as the whole area covered by all closed curves.

4.3 Component association

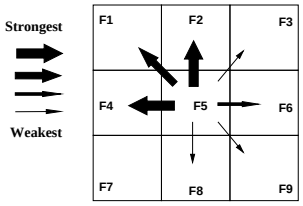
Having defined a component C_i, we define the component graph $G^C = (N^C, R^C)$, where the nodes are system components and the edges are *component association* links (or simply *association* links) between components. The association of the component C_i with the component C_j, denoted as *comp-assoc*(C_i, C_j) is defined as: *overall association-degrees of the entities in the domain of component* C_i *that also contained in component* C_j. The component association is a directed relation, hence, *comp-assoc*$(C_i, C_j) \neq$ *comp-assoc*(C_j, C_i). The steps for determining the component association are as follows:

Step 1:
Determining the *effective domain* of the component C_i onto the component C_j, i.e., $ED(C_i, C_j)$, by applying a *noise filtering heuristic*[3]. Generally, an entity domain D_k overlaps with many components. This filtering process restricts the effect of each domain D_k (whose main-node is in C_i) to a few components whose entities have non-trivial overlap with D_k. For example, in Figure 4(a), three entity domains from file F_5 have non-trivial overlap with the entities in file F_2 (highlighted with grey color) and generate the effective domain F_5 onto F_2, i.e., $ED(F_5, F_2)$.

[3]Because of space limitation the details are not discussed here.



(a) Component association among files



(b) Demonstration of the strength of the component association in part(a).

Figure 4. A system of nine files representing the association of *file 5* with its surrounding files.

Step 2:
Determining the component association between every pair of the components. The *comp-assoc*(C_i, C_j) is calculated as: *average of the total association degrees of every entity in* $ED(C_i, C_j)$[4]. Formally,

$$\textit{comp-assoc}\ (C_i, C_j) = \frac{\sum_{k=1}^{|ED(C_i,C_j)|} totalAssoc(e_k)}{|C_j|}$$

Where, $|ED(C_i, C_j)|$ is the number of the entities in $ED(C_i, C_j)$; $totalAssoc(e_k)$ is the sum of all association degrees for the k^{th} entity in $ED(C_i, C_j)$; and $|C_j|$ is the number of entities in component C_j. The unit of the *comp-assoc*(C_i, C_j) is *"association-degree per entity"* (abbreviated as APE).

The association of a component C_i on itself is defined as *comp-assoc*(C_i, C_i):

$$\textit{comp-assoc}\ (C_i, C_i) = \frac{\sum_{k=1}^{|ED(C_i,C_i)|} totalAssoc(e_k)}{|C_i|}$$

[4]Note that each node in *ED*(C_i, C_j) has different association degrees, if it is a member of different overlapped domains, as in Figure 4(a).

4.4 Association quantization

The above values for component association are distributed over a broad range that may be unmanageable. To quantify the values of component association, we use four ranges of values namely, *strong*, *medium*, *loose*, and *weak* (denoted as *strength of the association*). Figure 4(b) demonstrates a graphical quantization of the association values in Figure 4(a). In this example, *file 5* has strong-association to *files 2 and 4*, medium-association to *file 1*, and low-association to other files. The thickness of the arrows in Figure 4(b) can be viewed as different colors for the links among the files of a system within a graph visualization tool.

5 Architectural design evaluation model

In this section, we introduce an *architectural design evaluation model* in order to quantitatively evaluate the design of a software system based on its component interaction properties. The proposed model: i) provides measures of the component interactions based on abstract and meaningful data-/control-dependencies; ii) categorizes the overall architectural design of the system into different design properties; and iii) evaluates the modularity (quality) of alternative designs or architectures.

5.1 Association views

We generate three almost orthogonal association views for a system. Each view is generated by: i) maintaining the relations pertaining to the desired view in the entity relation database of the system and filtering out other relations; and ii) generating the component graph G^C of the system. Each resulting view is illustrated as a projection of the component graph based on a particular property. The views are defined as:

- *Control passing view* (*F-view*): representing the correlation among the system components based on function invocation, which is illustrated by the component graph G^C_F. The relation *"function calls function"* generates the *F-view*.
- *Data exchange view* (*T-view*): representing the correlation among the system components based on aggregate data types that are either passed as parameters between two functions or are referenced by a function. This view excludes any parameter passing with simple data types such as integer, real, boolean, and string, since they fail to show enough evidence of correlation between the functions. T-view is represented by the component graph G^C_T and is generated using the relation *"function uses types"*.

- *Data sharing view* (*V-view*): representing the correlation among the system components based on sharing the global variables by the functions. V-view is represented by the component graph G_V^C, and is generated using the the relation *"function uses variable"*, where the variable is global with simple or aggregate type.

The composition of these views are also feasible by considering the collection of relations in different views. However, this composition is not the superposition of the individual views since adding extra relations to each view may relate small associated groups of entities to generate larger groups. The composition of *T-view* and *V-view* generates the component graph G_{TV}^C, and the composition of all views generates the component graph G^C. The individual views can be used to obtain useful information about other design properties of the system as discussed below.

5.2 Design properties

In general, the architectural design decisions in developing a software system is affected by: i) corresponding domain of the system, i.e., existing a reference architecture; ii) employed design methodology, i.e., structured or object-oriented; iii) limitation of the employed tool or platform (e.g., memory limitation); and iv) software quality considerations (e.g., maintainability, modifiability). We consider three design properties for a system regardless of the adopted design methodology and its architectural style. We then relate these design properties to our proposed association views. In the following part, the terms "dominant", "medium", and "low" for describing a view, refer to the relative comparison of the values of the three association views in a system. The value for each association view is defined as: *average value of the component association per component* in that view (section 7.2 provides experimented values). The design properties that we consider include:

- *State sharing*: in this design property, the components (i.e., file, module, or subsystem) perform the desired operation of the system through accessing and modifying a number of global variables. In a system with such design property, *common coupling* is the dominant association among the components. This property is manifested by a large number of references from different system functions to these global variables while the other properties (mentioned below) are less visible. We distinguish such a property when a system has a dominant V-view and low T-view and F-view.

- *State passing*: in this design property, the system state is kept in data structures and the system operation is performed by changing and passing these data structures among different modules. In such systems, the coupling between components are mostly of the form *stamp* and *data* couplings. A system with such a property has a dominant T-view, medium F-view, and low V-view.

- *State encapsulating*: in this design property, the state of the system is encapsulated in the modules and the system task is performed by invoking different services to be performed by the modules using their own states. This property is known to be the best from the understandability and maintainability point of view. In such systems, the dominant couplings are *control passing* and *stamp coupling*. A system with such a property has a dominant F-view, medium T-view, and low V-view.

A hybrid system may use different design properties for different parts of the system. In Such cases, each view of the system demonstrate a density of association links in a different part of the system.

5.3 Modularity measurement

In section 4.3, we defined *comp-assoc*(C_i, C_j) and *comp-assoc*(C_i, C_i) for two components C_i and C_j. In this section, we define metrics to measure the quality of a system design or its decomposition, in terms of modularity. Each system component C_i can be assessed by three metrics such as: i) *selfAssoc-degree*(C_i), to measure the overall association of C_i on itself; ii) *assocOn-degree*(C_i), to measure the overall association of C_i on the rest of the system; and iii) *assocBy-degree*(C_i), to measure the association of the rest of the system on C_i. The analogy of these metrics to coupling and cohesion metrics are *cohesion-degree*, *couplesTo-degree*, and *coupledBy-degree*, respectively. Formally:

$$\textit{selfAssoc-degree}\ (C_i) = \textit{comp-assoc}\ (C_i, C_i)$$

$$\textit{assocOn-degree}\ (C_i) = \frac{n_{ij} \times \sum_{k=1}^{n_{ij}} \textit{comp-assoc}(C_i, C_{j_k}) \times |C_{j_k}|}{\sum_{k=1}^{n_{ij}} |C_{j_k}|}$$

$$\textit{assocBy-degree}(C_i) = \sum_{k=1}^{n_{ji}} \textit{comp-assoc}(C_{j_k}, C_i)$$

Where, n_{ij} (n_{ji}) is the number of components C_{j_k} that are the sink (source) of the association links from (to) component C_i in graph G^C. The unit for these formulas is *"association-degree per entity"* (APE).

An explanation for the *assocOn-degree*(C_i) formula is necessary. This formula first computes the average association of C_i on the other components linked to it (by merging all linked components into one component), and then multiplies this average to the number of the linked components. This method compensates for the size variation of the linked components and produces a uniform value regardless

of how the group of the linked components are divided into n components with different sizes. An alternative formula for *assocOn-degree*(C_i) may directly add all component associations of the linked components which is sensitive to the size variation of the linked components.

Having defined the above metrics for each system component, the *modularity-degree* of the whole system is defined as: *the average of difference between "self-association" and "association on others" for a component in the whole system.* Formally:

$$\textit{modularity-degree}(G^C) = \frac{\sum_{i=1}^{|N^C|}[\textit{selfAssoc-degree}(C_i) - \text{assocOn-degree}(C_i)]}{|N^C|}$$

The above metrics allow us to assess the design quality of a software system as well as the quality of the system decomposition based on the component association. In the following sections, the experimental results of this evaluation model are discussed.

The value for the modularity-degree can be negative or positive based on the weights of the overall self-association and association on others for the whole system files. In order to normalize the values for the modularity-degrees of different systems, we need a *reference system* to provide a reference modularity value. However, since we have not experimented with a large number of systems yet, for this paper we present a modularity comparison of a group of systems with each other.

6 Software architecture recovery tool

We have implemented a reverse engineering tool (Alborz), as a user assistant, to recover the architecture of a software system as cohesive components. The Alborz tool has been built in the Refine re-engineering environment [16] and uses the built-in parsers to parse the software systems, and the built-in parser generator to design a proprietary language called *Architectural Query Language* (AQL). The tool provides two complementary techniques for architectural recovery based on:

- *Component association technique*, where the tool provides a quantized component graph of the system components and association strengths, and the user visualizes the graph using a graph visualizer tool and clusters the components into larger subsystems. The *manual-blocks* of an AQL query are used to define the resulting components for the tool to be further analyzed. The result of the system decomposition into subsystems also provides a graph-based architectural pattern for the system to be used by the second technique.
- *Pattern matching technique*, where the user defines a graph-based architectural pattern of the system components and their interactions based on: domain knowledge, system documents, or component association properties. In an iterative recovery process, the user constraints the architectural pattern and the tool provides a decomposition of the system entities into components that satisfy the constraints. The *query-blocks* of an AQL query are used to define a constraint architectural pattern [18].

In both techniques, the tool provides metrics to assess the modularity quality of the software system and its decomposition into subsystems. The analysis techniques discussed in this paper are used as the first step in an iterative *scenario* for architectural recovery in each technique. These analyses (complemented with the system documentation) provide enough insight into the design of the system under analysis to enable the user to decide on the results at each iteration and the way to proceed with the next iteration.

The input to the Alborz tool is the entities and relationships of the software system which are extracted from either of the two sources: i) AST of the software system generated by the Refine's built-in parser; or ii) RSF format files generated by the Rigi parser. The tool provides the result of the architectural analysis into two forms: i) HTML pages for the recovered components, tool generated metrics, and source code, to be visualized by a Web browser such as Netscape; and ii) graphs of boxes and arrows to be visualized by the Rigi tool, 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.

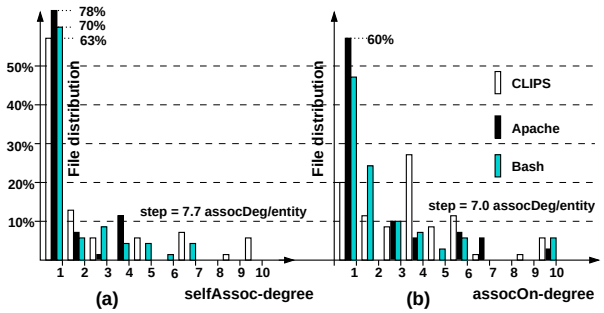
7 Experiments

In this section, we apply the proposed evaluation technique on five software systems. First, the modularity quality of the software systems are discussed, and next, the association views of a system are presented and the overall design properties of the five systems are compared.

Our experimentation platform consists of a Sun Ultra 10 (440MHZ, 256M memory, 512M swap disk). The experiments have been performed on systems written in C, including: CLIPS (expert system builder) [2], Xfig (drawing tool) [1], BASH (Unix shell) [3], Apache (Web server) [4], and Weltab (election system).

7.1 Modularity assessment

The tool provides three metrics *selfAssoc-degree*(C_i), *assocOn-degree*(C_i), and *assocBy-degree*(C_i) for individual files of the system, as well as the distribution of the



System (No. of files)	CLIPS (44)	Apache (42)	Bash (47)	Xfig (98)	Weltab (38)
Modularity degree	-11.6	-8.75	-5.58	-15.0	-2.97

(c)

Figure 5. (a) and (b) Comparison of file distribution versus component-association values for CLIPS, Apache, and Bash. (c) The modularity-degrees of five experimented systems.

files versus these metrics. The diagrams in Figures 5(a) and (b) compare the file distributions of three systems CLIPS, Apache, and Bash in one scale, versus two of the above metrics. The modularity-degrees of CLIPS, Apache, and Bash are: *-11.6*, *-8.75*, and *-5.58*, respectively.

In all three systems, on average the degree of association of a file to other files is higher than the degree of its self-association, causing a negative value for the modularity-degree of each system. The higher value for modularity-degree means more modular system. In comparison, the CLIPS files have higher value of assocOn-degree than other two systems which makes it less modular than the others. The Bash system is the most modular system among the others.

The table in Figure 5(c) presents a comparison of the modularity-degrees for five experimented systems. The Xfig system has the lowest modularity-degree. As will be discussed in the next section, the design of the Xfig system is towards using a large number of global variables (1356 variables), making the system less modular. The Weltab system has the highest modularity-degree among others. The component graph of the Weltab system is shown in Figure 6. The system contains of 38 files: i) two library files, i.e., *wellib.c* and *baselib.c* each with 21 functions; and ii) 36 other files, each with one to four long functions. The library file *wellib.c* is highly cohesive, and all other 36 files use the services provided by these two library files and have almost no association links between each other. This is verified by investigating the Weltab's source

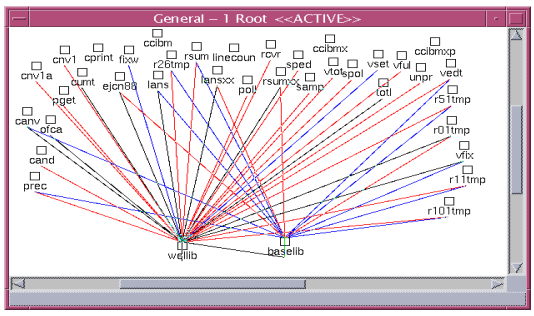


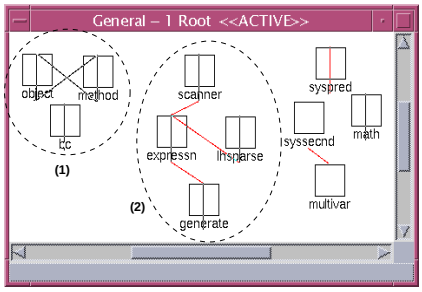
Figure 6. The component graph of the Weltab system, with two library files as the core of component associations.

code. The uncommon design of the Weltab system makes it modular since each file depends on only two library files which themselves are cohesive. Therefore, the change propagation is restricted to the modified file and two library files.

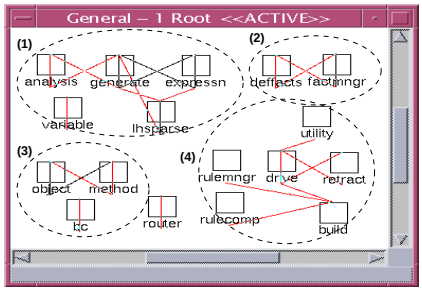
7.2 Design properties

In the view-based architectural design evaluation model, discussed in section 5, the software system is considered from almost orthogonal views which are used to categorize the overall design of the system into a set of proposed design properties. Figure 7 illustrates three association views (parts a,b,c) and the compositional view (part d) of the CLIPS system. In each part, the files are grouped based on the strength of the association links between the files. In the compositional view, this grouping of the files produces four subsystems for CLIPS.

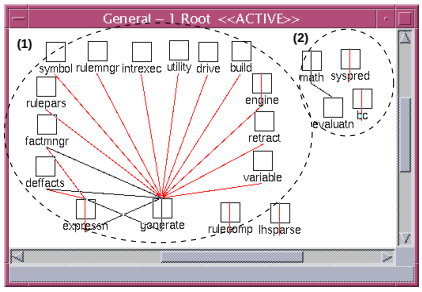
In order to highlight the differences in each view of Figure 7, only the files with strong and medium association strengths are shown. Except few cases, each view contains different sets of files and links which indicate the dominant property in each group of files. For example, group 1 in *F-view* is the same as group 3 in *T-view* which means this group of files are highly correlated through passing of aggregate data structures and also control passing. This subsystem is called *object* and demonstrates the *state encapsulation* property in its design. An investigation of different views reveals the contribution of each view in the compositional view (FTV-view). For example, the object subsystem discussed above is seen as the subsystem 2 in FTV-view; group 2 in F-view is seen as a part of subsystem 1 in FTV-view; group 1 in V-view contributes in generating the subsystem 1 in FTV-view, and represents the dominant coupling, i.e., common coupling, in the whole system. Therefore, comparison of different views with the compositional



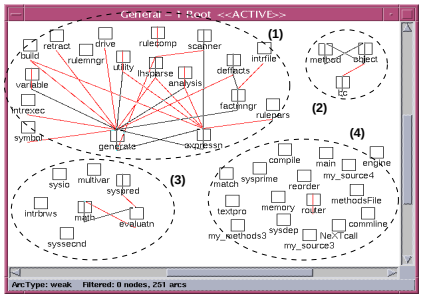
(a) Control passing view (F-view).



(b) Data exchange view (T-view)



(c) Data sharing view (V-view)



(d) Component graph G^C of the whole system (FTV-view)

Figure 7. The projection of the component graph G^C based on particular data-/control-dependencies produces almost orthogonal association views.

Target system (No. of files)	Min-support	F-view	T-view	V-view	FTV-view	Design property
CLIPS (44)	3	1.87	2.73	2.91	4.18	State sharing & State passing
Apache (42)	2	1.63	0.28	1.0	1.81	State encapsulation
Bash (47)	3	1.58	1.98	1.28	2.59	State passing
Xfig (98)	13	1.32	1.43	3.03	3.21	State sharing
Weltab (38)	27	1.46	N/A	2.4	3.71	State sharing

Figure 8. The evaluation table for comparing association views and design properties of five systems.

view provides additional knowledge about the design property of different parts of the system under investigation.

In Figure 8, the evaluation table for all five experimented systems is shown. In this table, the average of association value for each association view is used to assess the affect of that view in the overall design property of the system. In each case, one or two views are dominant which determine the overall design property. For the CLIPS system, the V-view and T-view are dominant, therefore CLIPS design is categorized as using both state sharing and state passing properties. As was mentioned earlier, the *object* subsystem of CLIPS has been designed with state encapsulation in mind, therefore, CLIPS has a hybrid design property. The association views for the Apache system does not completely match with our categorization, but it is toward state encapsulation property. For the Bash system, the values for views match with our definition of state passing property. Finally, both Xfig and Weltab systems have dominant V-views which categorize them as state sharing design. It is seen that the value of the composition view (FTV-view) for each system is higher than each individual view. This is because combining the relations of the three views produces larger associated groups of the system entities, hence the average association value increases.

The *min-support* column in Figure 8 represents the employed minimum support for generating the data mining associations. Although, the ideal case is to use the minimum support 2, but the size of the generated frequent itemsets explode in systems with many groups of entities that are highly associated. In such cases, we have to increase the minimum support in order to make the algorithm tractable. The case of Weltab system with minimum support 27 is rare and indicates the existence of very large associated groups of functions in that system.

8 Conclusion

In this paper, we proposed an architectural design evaluation model to assess the design properties of a system based on the association-views of the system. We also evaluated the modularity-degree of the software system. The model is based on the notion of *component association* as a generalization of the coupling and cohesion metrics at the architectural level of a system. The proposed system analysis and evaluation techniques assist the user to obtain insight into the system under analysis and provides means to assess the result of the recovery which is essential in performing a reverse engineering task.

The techniques are integrated into a tool which conveniently presents the results of the analysis as HTML pages and graphs to be visualized by Rigi tool. A number of experiments with five middle size systems indicate the accuracy and scalability of the approach, and the usefulness of the Alborz tool in producing meaningful metrics. This work has been performed within the framework of the Consortium for Software Engineering Research and in cooperation with IBM Toronto Laboratory, Center for Advanced Studies.

Acknowledgment: I would like to thank Dr. Kostas Kontogiannis for his fruitful suggestions to enhance this paper.

References

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

[2] CLIPS expert system builder, Web site, URL = http://www.ghg.net/clips/CLIPS.html.

[3] Bash Unix shell, Web site, URL = http://www.delorie.com/gnu/docs/bash/.

[4] Apache HTTP Server, Web site, URL = http://www.apache.org/httpd.html.

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

[6] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

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

[8] R. Koschke. An incremental semi-automatic method for component recovery. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 256–267, October 1999.

[9] T. Kunz and J. P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527, June 1995.

[10] C. Lindig and G. 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.

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

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

[13] V. B. Misic. Coherence equals cohesion-or does it? In *Proceedings of the Seventh Conference on Asia-Pacific Software Engineering*, pages 465–469, December 2000.

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

[15] R. S. Pressman. *Software Engineering, A Practitioner Approach*. McGraw-Hill, third edition, 1992.

[16] Reasoning Systems Inc., Palo Alto, CA. *Refine User's Guide*, version 3.0 edition, May 1990.

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

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