Architectural Design Recovery using Data Mining Techniques *

Kamran Sartipi[1] Kostas Kontogiannis[2] Farhad Mavaddat[1]

University of Waterloo
Dept. of Computer Science[1] and,
Dept. of Electrical &
Computer Engineering[2]
Waterloo, ON. N2L 3G1
Canada

Abstract

This paper presents a technique for recovering the high level design of legacy software systems according to user defined architectural plans. Architectural plans are represented using a description language and specify system components and their interfaces. Such descriptions are viewed as queries that are applied on a large data base which stores information extracted from the source code of the subject legacy system. Data mining techniques and a modified branch and bound search algorithm are used to control the matching process, by which the query is satisfied and query variables are instantiated. The matching process allows the alternative results to be ranked according to data mining associations and clustering techniques and, finally, be presented to the user.

1 Introduction

Software maintenance constitutes a major part of the software life-cycle. Most maintenance tasks require a decomposition of the legacy system into modules and functional units.

One approach to architectural design recovery is to partition the legacy system using clustering, data-flow and control-flow analysis techniques [16]. Another approach is based on user defined constraints that need to be satisfied [28], therefore, architectural recovery becomes a Constraint Satisfaction Problem (CSP). We propose an alternative approach, where architectural design recovery is based on design descriptions that are provided by the user in the form of queries. We call this formalism *Architectural Query Language* (AQL).

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

In the proposed approach, the architectural design recovery process consists of three phases. In the first phase, the source code is represented at a higher level of abstraction using Abstract Syntax Trees and entity-relationship tuples. In the second phase, the user defines queries in AQL based on a hypothesis about the system's assumed architecture (i.e., *conceptual architecture*). Finally in the third phase, a pattern matching engine finds the closest match between the query-based specification and a collection of source code components, generating thus a *concrete* architecture [16] from the AQL query. In this sense, the AQL query provides a description of the conceptual architecture and the instantiated query provides the corresponding concrete architecture. The query allows us to obtain an optimal arrangement of the functions, types, and variables in the modules that conform to the user's view of the conceptual architecture. The optimal arrangement is obtained with respect to the evidences gathered from the source code using data mining and clustering techniques. The concrete architecture that results from the matching process and the user provided AQL query can be thought of as a form of *goal-directed* clustering.

The proposed approach focuses on facilitating partial matching, a situation that is frequent in practice and has been addressed in a framework of uncertainty reasoning. It differs from other approaches in the area of constraint satisfaction [28], in the sense that the matching process is guided by the properties of the subject system, as opposed to satisfying constraints. As a result, instantiating AQL query variables becomes a problem of maximizing similarity values as opposed to satisfying constraints.

Considering the size of the *search space* for the pattern matching engine when a large system is involved, the scalability of the approach is a fundamental requirement. In order to limit the search space and speed-up the matching

process, we use data mining techniques and a variation of the branch and bound search algorithm. In general, we assume that the user relies on the domain knowledge to compose the queries.

2 Related work

The following approaches are related to our approach. The Murphy's reflexion model [22] allows the user to test a high level conceptual model of the system against the existing high level relations between the system's modules. In our approach the user describes a high level conceptual model of the system and the tool provides a decomposition of the system into interacting modules. Some clustering techniques also provide modularization of a software system based on file interactions and partitioning methods [21]. Specialized queries (recognizers) for extracting particular properties from the source code are presented in [12, 15]. In [6] a tool for code segmentation and clustering using dependency and data flow analysis is discussed. Holt [16] presents a system for manipulating the source code abstractions and entity-relationship diagrams using Tarski algebra. The system recovers aggregations and design abstractions in large legacy systems. In [8] a clustering approach based on data mining techniques is presented. Lague et. al present a methodology for recovering the architecture of the layered systems [19]. The methodology focuses on the examination of interfaces between different system entities.

In this work, we use the notion of Architecture Query Language (AQL) which is a direct extension of Architectural Description Languages (ADL) as discussed in: Unicon [25], Rapide [20] and, ACME [13].

3 Architectural design recovery

We consider four fundamental views for software architecture namely, *structure, behavior, environment*, and *domain-specific* [2]. The notion of views has been discussed extensively in the literature [18]. In a broad sense, views are the result of applying *separation of concerns* on a design in order to classify the related knowledge about the design into more understandable and manageable forms.

In this paper we focus on the structural view of the architecture. The structural view covers all building blocks and interconnections that *statically* describe the architecture of a software system. It consists of *static* features[1] and *snapshot* features[2]. In particular, given a legacy system which is represented as an unstructured or poorly structured collection of files, functions, and data type declarations (due to prolonged maintenance and evolution), we are interested in obtaining a decomposition of the legacy system into a set of structured modules.

We have developed a tool for structural recovery as well as restructuring a legacy system using a query description language (AQL). The matching process is an optimization task in which the maximum value of a score function is sought at every step of the process. We use data mining and clustering techniques to evaluate the score function.

Within this context, a module is defined as a conceptual and arbitrary large collection of consecutive source code fragments with an aggregate name [14, 24].

A *module* (e.g., *M*) is considered to be a collection of functions *F*, data types *T* , and variables *V* (including both the target system entities and the library items) that constitute a set of tuples of the form `<Module Relationship Entity>`. We can represent these tuples using the relations[3] *contain, import*, and *export* that constitute the whole architecture. In the above tuple, `Module` is "*module moduleName* ", `Relationship` is *contain, import*, or *export*, and `Entity` is a *typed name* that refers to a *Function definition, Data type*, or *Global variable*.

More formally, let $\mathcal{F}, \mathcal{T}$, and $\mathcal{V}$ be the sets for all functions, all data types, and all global variables, that appear in a given legacy system or its associated library. We consider a module as a triple $M = \langle F, T, V \rangle$ where:

$$F = \{f \mid Function(f)^{4} \wedge ((\langle M, f \rangle \in contain \vee \langle M, f \rangle \in export) \oplus^{5} \langle M, f \rangle \in import)\} \subseteq \mathcal{F},$$

$$T = \{t \mid DataType(t) \wedge ((\langle M, t \rangle \in contain \vee \langle M, t \rangle \in export) \oplus \langle M, t \rangle \in import)\} \subseteq \mathcal{T},$$

$$V = \{v \mid Variable(v) \wedge ((\langle M, v \rangle \in contain \vee \langle M, v \rangle \in export) \oplus \langle M, v \rangle \in import)\} \subseteq \mathcal{V},$$

$$\langle M, e \rangle \in export \Rightarrow \langle M, e \rangle \in contain.$$

The semantics of the relations *contain, import*, and *export* are those presented in the standard Software Engineering literature [14]. A module contains functions, types, and global variables, which can be exported to other modules or (if not contained) can be imported from other modules.

[1] The "static" features are information that can be extracted by statically analyzing the source program.

[2] The "snapshot" features are information that can be detected statically by interrupting a running program and registering the program's context and state.

[3] The term "relation" denotes to a set of pairs of elements.

[4] *Function(f), DataType(t),* and *Variable(v)* are predicates that recognize the entity-type of an entity.

[5] $\oplus$ denotes the XOR logical operation.

Having defined the notion of a module, we can define an architecture A to be a decomposition of a system into *n* modules where, $A = \{m_i = \langle F_i, T_i, V_i \rangle \mid i \in [1..n]\}$.

However, the collection of entities contained in the modules that constitute an architecture A may not cover all the entities of the legacy system. This case occurs when some entities can not be grouped into an identifiable module [8] which leads us to the problem of orphan adoption discussed in [26].

Within this research framework the issues to be addressed include:

- A schema for modeling a database that contains information related to the system under analysis.
- A formalism for representing an abstract architectural design in the form of queries.
- A tractable grouping methodology (based on data mining) that reveals strong associations between elements of the software system stored in the database. The strength of the associations is used as a mechanism to drive the pattern matching process.
- An approximate pattern matching engine that allows a legacy system to be decomposed based on the given user defined queries (patterns).
- An efficient method of presenting the results to the user.

The above topics will be discussed in more details in the following sections.

3.1 Data mining

Data mining or Knowledge Discovery in Databases (KDD), refers to a collection of algorithms for discovering interesting and non-trivial relationships among data in large databases [10]. Most data mining algorithms are based on the concept of database *transactions*[6] and their *items* that correspond to *market baskets*. In our approach, each transaction is a function definition F_t from the software system under analysis, and the transaction items are the system functions, data types, and global variables (Figure 1) that are called or used in any form by F_t. A more detailed discussion on data mining concepts is presented in the following section. A transaction contains different kinds of items. In this context, the quantity of items of the same kind in a transaction is not considered. Interesting relationships may be discovered, using data mining, among groups of items

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

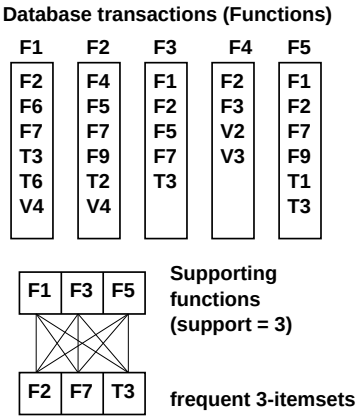

Figure 1. An application of the "database transaction" notion in Reverse Engineering domain. The functions F1, F3, and F5 all call or use the functions F2, F7, and data type T3.

in transactions (*association rules*) [4], among sequences of groups of items in transactions (*sequential patterns*) [5], or among the time of occurances of transactions (*time-series clustering*) [3].

3.2 Frequent itemsets

Interesting properties of data in a database, namely association rules, are extracted from *frequent itemsets*. A *k-itemset* is a set with cardinality $k > 0$. A frequent itemset is an itemset whose elements are contained in every member of a group of supporting transactions (i.e., supporting functions, Figure 1). The cardinality of this group of transactions is greater than a user-defined threshold called *min-support*. The frequent itemsets are generated by the *Apriori* algorithm [4]. In Reverse Engineering domain the example entities are file, function, and data type, and the example relationships are fetch, store, define, and call. Figure 2 demonstrates how the *containment* relationship between a transaction (basket) and its items in data mining domain can be extended to the relationship between a *container* and a set of *item-operation* in Reverse Engineering domain. An item-operation is a collection of an entity and the *operation* on that entity. This collection can be treated as an item in a transaction. The first example of the table is interpreted as: a function (file) consists of a set of *call-to-function* (-file).

Based on the above discussion, we can consider that a function (as a transaction) consist-of "call-to-function F-xx", "use-variable V-xx", and "use-type T-xx" (as items of the transaction). The *use* relationship is interpreted as fetching or storing to a global variable, or data type inside a function. A collection of frequent i-itemsets ($i \in \{1..k\}$)

	Entity	Relation	Entity	
Data mining	a transaction	consists of	item	
Reverse engineering	a container	consists of	item-operation	
	file / function	consists of	call to	files / functions
	file / function		use (read / write)	files / types / vars
	file / function		write to / read from	pipes
	file / function		send to / receive from	sockets
	file		import / export	functions / types / vars

Figure 2. The conversion of various relationships in Reverse Engineering paradigm into the containment relationship of a database transaction.

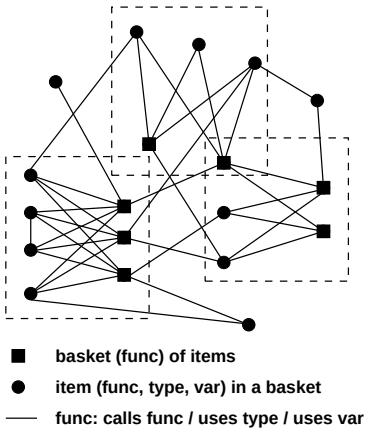

Figure 3. A bi-partite sub-graph representation of the frequent itemsets produced by the Apriori algorithm.

along with the container functions (transactions) are generated and stored to be further processed.

3.3 Database for matching process

We use the Refine[7] re-engineering tool [23] to parse the target system and populate an object-oriented database with the facts that are extracted from the target system. Using the Refine environment, we build a database containing tables that correspond to the relations (tuples) of the form: `<function calls function>`, `<function uses type>`, and `<function uses variable>`. Data mining and *clustering* techniques [9, 17, 27] are applied on this database to extract strong associations between the entities. The data mining Apriori algorithm reveals the existing association among entities in the form of frequent itemsets. In a frequent k-itemset, k can be viewed as the *strength* of the association among every pair of items in the collection of the itemset and its supporting functions. A high strength of association among a group of system entities indicates high cohesion among those entities and qualifies them as candidates to be put in the same module. A sample of the frequent itemsets is shown below:

```
1 <[V-3 T-42 T-44 T-58] [F-83 F-176 F-646 F-647] 4>
2 <[V-3 T-43 T-44 T-58] [F-83 F-647] 2>
3 <[V-3 F-478 F-649 F-719] [F-647 F-648] 2>
4 <[V-4 T-41 T-42 T-44] [F-83 F-647 F-648] 3>
5 <[V-30 F-552 F-553 F-567] [F-547 F-548] 2>
```

Each line is a record in the database consisting of an itemset (left), followed by the transactions (baskets), and the itemset support (i.e., the number of transactions). The target system's entities have been encoded into an identification letter (e.g. **V** for variable, **T** for type, **F** for function), and an id-number that uniquely identifies an entity. For example, the first line of the sample data above is interpreted as: *each* of the functions F-83, F-176, F-646, and F-647 uses *all* variable and data-types denoted by V-3, T-42, T-44, and T-58. These records are part of the *frequent 4-itemsets* (i.e., 4 items in an itemset).

Figure 3 illustrates a bi-partite sub-graph representation of the frequent itemsets. Three complete bi-partite sub-graphs are shown inside the dashed boxes. These sub-graphs signify the high cohesion among the group of involving functions. Therefore, it is promising to assume each sub-graph as a potential skeleton of a module.

3.4 The role of data mining in the recovery process

In this section, we elaborate on the significance of the data mining technique Apriori in our approach:

- The frequent itemsets, discussed in the previous section, are used to generate a collection of entities that can be considered as the candidates to be contained in a module, given a *seed* for that module. We call this collection a *domain*. To generate a domain, we collect all those entities that co-exist with an entity *s* (we call it *main-seed* s) in any single frequent itemset, along with the entity's highest association value with the main-seed s. The domain of s is denoted by $D(s)$. Below, the set of entities in $D(s)$ (without the association values) are defined:

  $$D(s) = \{d \mid \forall k \in [1..|F|],\ \{d, s\} \subset (F_k.I \cup F_k.T)\}$$

  where, F is the whole collection of frequent itemsets, F_k is a single itemset record, and I and T are the itemset and its supporting transactions. For example, if the

[7] Refine is a trademark of Reasoning Systems Inc.

whole frequent itemsets F in the system are those 5 records that we presented above, then the domain of function F-83 is as follows:

```
 D(F-83):
{<V-3 4> <T-42 4> <T-44 4> <T-58 4>
 <F-176 4> <F-646 4> <F-647 4> <V-4 3>
 <T-41 3> <F-648 3> <T-43 2>}
```

The collected domains are the basis for grouping the entities into modules.

- In a pre-processing phase, a domain analysis algorithm examines the association values of the entities in the domain of each potential main-seed in order to come up with the best candidate domain for the variables of each module in the query. The matching process then uses the suggested domain for each module as its search space to instantiate the query variables for that module.

- The association value of each entity in a domain can be used as an input to a closeness-score valuation function (please see section 5.2) for selecting the best *n* entities to be contained in a given module (where *n* is determined by the number of variables in the query part that corresponds to the module).

Therefore, a module is built around a core entity, i.e, the *main-seed*, of that module, and data mining provides a restricted and highly associated domain of values for the variables in that module, as well as a criterion for grouping the entities in that module.

4 A system level query language

In this section we present an overview of the Architectural Query Language (AQL) which is used for describing (not specifying) the conceptual architecture of a legacy system. The AQL allows for:

- decomposing the program representation into modules with inter-/intra-module relationships,

- representing alternative design views (i.e. structural or behavioral), and

- abstracting away the target system's syntactical and implementation variations.

The syntax of AQL encourages a structured description of the architecture for a part or the whole system. A typical AQL query is illustrated below:

```
BEGIN-AQL
MODULE: M1
   MAIN-SEED:    func initialize()
   IMPORTS:
     FUNCTIONS:  func $IF(0..2), func ?F1
     TYPES:      type $IT(0..1)
     VARIABLES:  var  $IV(0..1)
   EXPORTS:
     FUNCTIONS:  func $EF(0..2), func ?F2()
     TYPES:      type $ET(0..2), type ?T1
     VARIABLES:  var  $EV(0..1)
   CONTAINS:
     FUNCTIONS:  func $CF(2..16),
                 func initialize()
     TYPES:      type $CT(1..5)
     VARIABLES:  var  $CV(0..8)
END-ENTITY

MODULE: M2
   MAIN-SEED:    func deal_card()
   IMPORTS:
     FUNCTIONS:  func $IF(0..2), func ?F2()
     TYPES:      type $IT(0..2), type ?T1
     VARIABLES:  var  $IV(0..1)
   EXPORTS:
     FUNCTIONS:  func $EF(0..2), func ?F1
     TYPES:      type $ET(0..2)
     VARIABLES:  var  $EV(0..1)
   CONTAINS:
     FUNCTIONS:  func $CF(6..10),
                 func deal_card(),
                 func hit_player_hand_1(),
                 func hit_player_hand_2()
     TYPES:      type $CT(0..2)
     VARIABLES:  var  $CV(0..2)
END AQL
```

The prefixes "$" and "?" represent a multiple-valued and a single-valued placeholders, respectively. For example $CF(6..10) denotes a multi-valued placeholder that can be instantiated by *minimum* 6 and *maximum 10* functions that are contained in a module [8].

Single-valued placeholders, with the same name in different parts of a query, can only be instantiated with a single entity. The matching process provides a substitution σ, which binds these AQL placeholders with actual entities of the legacy system. When *all* placeholders in the query have been instantiated, i.e., bound to values (even by a NULL binding), a concrete system architecture is generated (as opposed to the abstract architecture defined by the AQL query).

5 Search and control

In traditional approaches to program understanding, a top-level control mechanism selects the program parts to be compared against the given pattern in query. In recent approaches to Reverse Engineering, the user-input is an important factor in guiding the whole recovery process [22, 11, 1, 7]. For this work, we use the branch and bound search algorithm for the matching process. This AI search

[8] We adopt a naming convention for the AQL variables, e.g., CF denotes to *contains functions*.

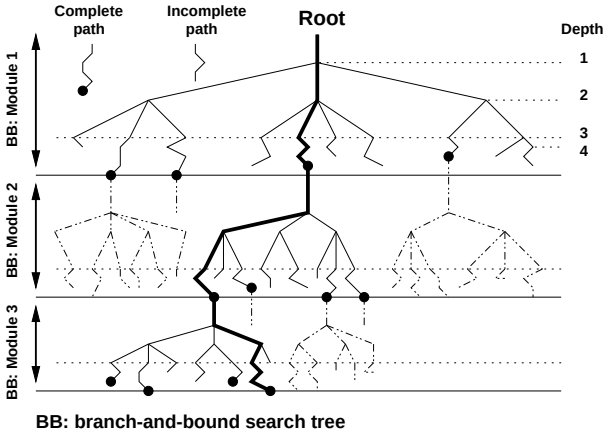

Figure 4. Three branch and bound search trees for instantiation of a three-module architecture described in an AQL query.

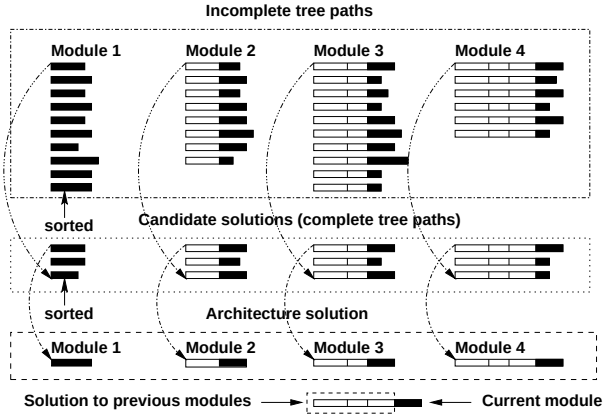

Figure 5. An implementation view of separate branch and bound search algorithms for incremental recovery of the modules.

technique is known to perform well in large search spaces since it explores the search tree based on the knowledge from the system.

5.1 Branch and bound

In a branch and bound algorithm a search tree with *incomplete paths* is built. At each step the algorithm expands an incomplete path with the highest score among all other incomplete paths. Upon expansion, new incomplete paths are generated and added to the previous ones. The procedure continues until a *complete path* which is an optimal solution is found. A valuation function allocates a score to each node of the branch and bound search tree to guide the search process. This general approach in most cases restricts the search space to a small subset of all tree paths, preventing the exponential complexity inherent to the searching problems.

More formally, given an AQL query $\mathcal{Q}$ containing a set of placeholders $\$X(1..n) = \{?X_1, ?X_2, ...?X_n\}$, the objective is to provide a substitution[9] $\sigma = \{?X_1/value_1, ?X_2/value_2, ... ?X_n/value_n\}$ that binds AQL query placeholders with actual source code entities. An internal node N_i in the search tree *T* is associated with a set of bindings $\mathcal{B}_i = \{?X_1/val_1, ..?X_i/val_i\}$ for those query placeholders instantiated so far. A leaf node N_l is associated with a set of bindings $\mathcal{B}_l = \{?X_1/val_1, ...?X_n/val_n\}$ that provide values for *all* placeholders appearing in the query. The bindings $\mathcal{B}_i$ and $\mathcal{B}_l$ are known as incomplete path and complete path, respectively. In this context, a search space is defined by all values that provide a possible binding to the query placeholders.

Properties of the search engine

The main-seed *s* of a module *M* is the first entity that instantiates a placeholder *?ph* inside a module. This entity determines the root of the corresponding search tree. The domain of *s* (i.e. D(s)) determines all potential entities that can be put in the module *M*. At each step of the branch and bound search, the number of paths that expand the incomplete path to an internal node N_i is equal to $|D(s)| - d$, where $|D(s)|$ is the domain size of s and d is the depth of node N_i. The max size of the module *M* (determined by the multi-valued placeholders) and the quality of the collected entities, determine the end of the search for each module. A score history, containing the evaluated scores at each depth of the search tree for each module, is maintained which reflects the quality of a recovered module.

Figure 4 demonstrates a sequence of branch and bound search trees that incrementally instantiate a concrete architecture consisting of three modules. A thick line from the root of the first tree to a leaf of the last tree represents an optimal architectural solution. In a macroscopic view, the architectural pattern of an AQL query is a graph of modules, connected via import/export links. A sequence of branch and bound searches provides solutions for individual modules and incrementally builds the concrete architecture. A link is a *shared entity* among two or more modules that is bound to a link placeholder (e.g., *?V1* or $?CF_i$ in the IMPORTS/EXPORTS parts of the linked modules.

In the restructuring phase, at every node of a search tree all single-valued placeholders between the current and pre-

[9]The case of multi-valued placeholder can be easily generalized to single-valued placeholders.

vious modules are examined for instantiation. The entities at each node, if are not used for single-valued link instantiations, are kept in a pool of entities to be used for future link instantiations. To ensure the correctness of this method, we always maintain the whole incomplete architecture *consistent* with respect to the bound links and the modules' *seeds* (i.e., main-seed and the other user-defined entities in the modules). By a consistent architecture, we mean that each architectural entity must be contained in only one module, and the shared entities must be correctly resolved.

The overall search control process uses a back tracking mechanism to allow alternative solutions for previous modules to be generated, should the solution for the current module fail to produce an architectural solution. In Figure 4 the paths in solid trees have been examined by the branch and bound search algorithm and some solutions have been found, whereas the dashed trees provide alternative solutions in backtracking phases.

In order to reduce the chance of being trapped in a local optimal solution, a number of complete paths (solutions) are collected for each module (*candidate solutions*), and the best candidate solution is selected for a module (Figure 5).

5.2 Score valuation

The score valuation of a new binding $?X_d/val_d$, occurring at some node N_i in depth d, is used by the branch and bound algorithm to expand the search tree. The branch and bound algorithm selects the best entity in the domain of main-seed (according to a score function) for the new binding. Some important criteria in modularization of a system include:

- providing low-coupling and high cohesion among entities of different modules,
- conforming with the user defined constraints, and
- collecting more entities in a module.

For evaluation of the score function we employ some notions from the clustering paradigm. A *similarity matrix* [17] is used to assess the closeness of a candidate item to a group of items in a module. This information is used in a *closeness-score function* and is evaluated at each node of the search tree.

The similarity matrix S is a symmetric matrix (i.e., $a_{ij} = a_{ji}$). Each matrix entry a_{ij} is a record of various similarity information between two system entities e_i and e_j (either function, type, or variable). The following categories are used to assess the closeness between entities:

- *Shared features*, which is based on the number of features (also called *property* or *attribute*) that exist for both entities. We use the *Jaccard* method ($Jaccard = \frac{A \cap B}{A \cup B}$, where A and B represent the sets of feature values for two entities) [9] to assess the closeness between two entities based on a single feature. For example, in measuring the closeness of two functions based on data type usage, A and B represent the sets of datatypes used by the two functions. The Jaccard similarity is obtained for each separate feature in a similarity matrix entry, and then an average closeness based on all features in that entry is obtained. The features to be considered include: *shared types*, *shared variables*, *shared child function*, and *shared parent function*. The only similarity feature between a function and a type/var is sharing scope of usage (i.e., parent function).
- *In-between relationships*, which are based on direct properties such as *call* between two functions, and *usage* between a function and a type/var. More complex relationships such as: *in-loop*, *call-depth*, and *dominance* relationships between two functions can also be considered.

The closeness-score function provides high score for an entity which has high "average value for shared features" and high "association level" with respect to the entities of a module. This guarantees a high-cohesion among a module's entities. Similarly, the closeness-score function provides high score for an entity that has high "relationships" with the members of a module. This guarantees a low-coupling among different modules.

The similarity matrix is generated in a pre-processing phase, hence, does not impose extra computation load during the search. The *group average* method (i.e., average value of closeness between an entity and each member of a group of entities) [9] is used to evaluate the closeness of an entity and a module's entities. The Jaccard and group average method are among the most popular techniques for similarity evaluation and clustering operation, respectively [9].

We use the following linear closeness-score function with empirical coefficients (between a candidate entity *E* and a group of entities *G* already in a module).

$$\textit{closeness-score (E, G)} = (2 * t.a.s + \frac{a.s}{10} + \frac{g.s}{20} - \frac{l.t.m}{g.s})$$

where $t.a.s$: total average similarity of the entity *E* with respect to all entities in the group *G*, $a.s$: association strength of the entity *E* with respect to the main-seed (from data mining), $g.s$: group size , and $l.t.m$: less than minimum number of bound multi-valued placeholders, obtained from the AQL query. The latter term signifies the demerit point if the module does not meet the user defined constrains for minimum number of entities in a module which are assigned through multi-valued placeholders.

6 Experimental results

We use two systems as the vehicles to conduct our experiments: i) the CLIPS system (C-Language Interface Processing System), a medium size (approx. 40 KLOC) rule-based system, with 734 functions, 59 aggregate data-types, and 163 global variables; ii) *TwentyOne*, a small game program with size 1.6 KLOC consisting of 3 files, 38 functions, and 16 global variables.

In this section, we investigate two groups of experiments. In the first group, we focus on the recovery and restructuring tasks of a target system into distinct modules. In the second group, we verify the precision, recall, and stability of the matching process, using an information retrieval framework.

Our experimentation platform consists of a Sun Ultra 10 (333MHZ, 256M memory) and the experiments are performed in a low CPU-load environment. It takes 4 minutes to parse the target system (CLIPS), using a parser written in Refine C, and to construct an annotated AST in the Refine's database. The Apriori algorithm requires approximately 20 minutes to build the frequent itemsets with support 2. The search time of the branch and bound algorithm is sensitive to the formulation of the score function presented in the previous section. We can break a long search for a module into incremental steps, hence, limiting the search time to less than 30 seconds for most cases.

6.1 Architectural recovery

6.1.1 Module recovery

The proper selection of the main-seeds[10] for the modules in an AQL query is the key point in architectural recovery of a system. The "closeness" among a group of entities that most likely belong to the same module is proportionally related to the association strength between each pair of entities in that group. This information is an important criterion for selecting distinct main-seeds when forming a query. A number of utility functions assist the user to select the main-seeds for the modules by analyzing the main-seeds' domain and comparing them with the already recovered functions in modules.

TwentyOne system

In this experiment we formed a query for recovering the architecture of a 1.6KLOC C program into five distinct modules. The result is shown in figure 6. In this experiment, using the techniques discussed earlier, we selected the following main-seeds: `initialize` (F-12), `deal_card` (F-18), `usage` (F-1), `player_wins` (F-24), and `get_insurance_for_player` (F-20) for the modules, respectively. A part of the query for this experiment has been shown as an AQL example in section 4. The output of the recovery for the first module is presented below:

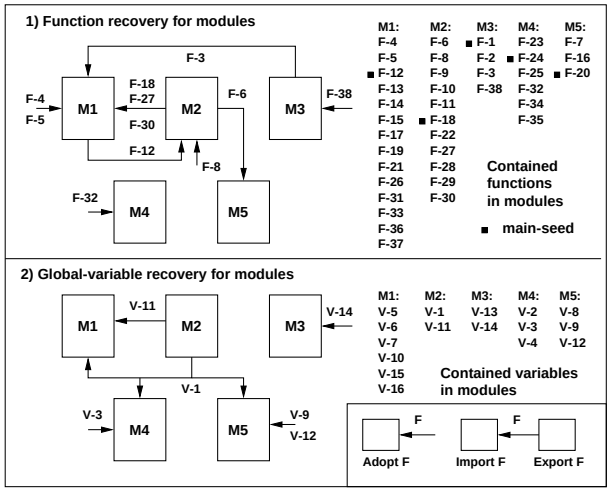

Figure 6. Modularization of a small program `TwentyOne` into five modules.

```
################  Module 1  ################
IMPORTS:
 F: ((Y1 F-3) (Y2 F-18) (Y3 F-27) (Y4 F-30))
 V: ((Y1 V-11) (Y2 V-1))
EXPORTS:
 F: ((Y1 F-12))
 V: (NIL)
CONTAINS:
 F: ((X1 F-5) (X2 F-4) (Y1 F-14) (Y2 F-13)
    (Y3 F-15) (Y4 F-17) (Y5 F-31) (Y6 F-36)
    (Y7 F-21) (Y8 F-33) (Y9 F-26) (Y10 F-37)
    (Y11 F-19) (Y12 F-12))
 V: ((Y1 V-6) (Y2 V-10) (Y3 V-16) (Y4 V-15)
    (Y5 V-5) (Y6 V-7))
    -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
 File: PLAY (V-16 V-15 F-12 F-19 F-37 F-26
       F-33 F-21 F-36 F-31 F-17 F-15 F-13
       F-14 F-18 F-27 F-30)
        ------------------------------------
 File: UTIL  (F-4 F-5)
        ------------------------------------
 File: MAIN  (F-3 V-7 V-5 V-10 V-6 V-11 V-1)
```

The Xs and Ys in the recovered parts are the instantiation of the multi-valued placeholders (e.g., $CF(2 ..16) generates 2 Xs and 14 Ys). The distribution of the recovered entities in the three files `main.c, play.c` and `util.c` are also shown. Figure 6 illustrate the shared and adopted entities among modules. The process is incremental in terms of the entity-type recovery, i.e., first

[10] Refer to section 3.4.

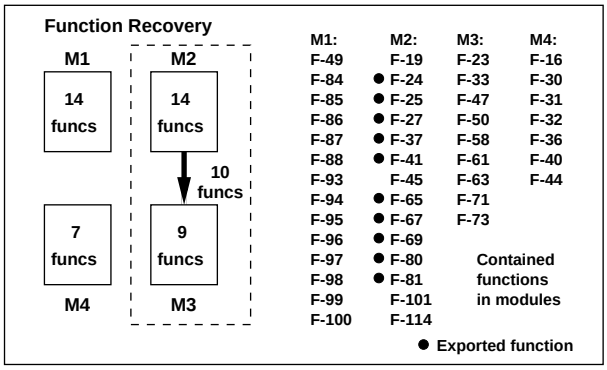

Figure 7. Modularization of the three files of the CLIPS system into four modules.

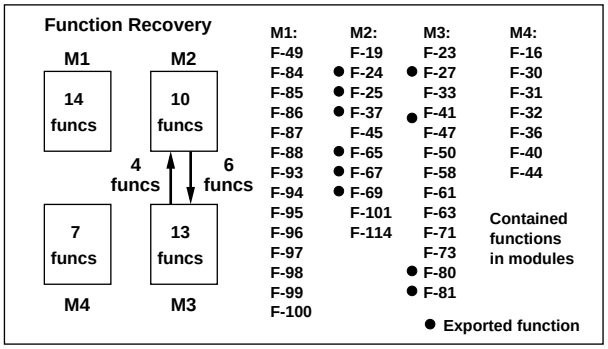

Figure 8. Restructuring of the recovered modules M2 and M3 in Figure 7.

functions and then global variables have been recovered.

CLIPS system

In the second experiment, a subsystem of the CLIPS, i.e., files `bc.c, object.c`, and `method.c` are considered for modularization. The main-seed domain analysis for these files showed that on average over 90% of the domain of each function (if it is considered as a main-seed) with high average association value is covered by the same three files and two other files (i.e., `factmngr.c` and `evaluatn.c`). Therefore we need to focus on only five files for our experiment as opposed to 46 files of the CLIPS. Figure 7 illustrates the result of experiment with four modules. The first module consists of 76% of the functions in the file `bc.c` with no shared entities with other modules. The second and third modules have 10 functions in common which indicates that these two modules are good candidates to join and make one module. The fourth module recovered the functions from the file `object.c`. An overall observation is that the file `bc.c` itself is a module but the other two files can be decomposed into two or more modules.

6.1.2 Module restructuring

In the recovery phase, discussed above, the target system is modularized according to the property of the system entities and their interaction. However, the recovered modules may produce an ill-formed modular system in terms of uneven module sizes, or unbalanced inter-module interactions. The proposed methodology allows the user to refine the AQL query and impose extra constraints on the recovery process. This causes the search algorithm to allocate high score values to those modules whose single-valued links have been instantiated and therefore, producing better modularization results.

For example, in the experiment above we obtained four modules out of three source files. The high interaction among the second and third module suggested that we combine those two modules into one module with a total of 23 functions. Another possibility is to maintain the two modules separate and impose the search algorithm to reconfigure the imported/exported entitles based on the single-valued query links, and then assign the remained shared entities as before. To do this we define two single-valued query links (see below) between the modules in the opposite direction of those shown in Figure 7 and run the query.

```
MODULE: M2
   IMPORTS:
     FUNCTIONS:     func $IF(0 .. 10),
                    func ?F1(),  func ?F2()
MODULE: M3
   EXPORTS:
     FUNCTIONS:     func $EF(0 .. 10),
                    func ?F1(),  func ?F2()
```

In the result illustrated in Figure 8, the single-valued links have been instantiated with two functions (i.e., F-80 and F-81). Also, two other functions (i.e., F-27 and F-41) have been moved from the second module to the third module. This replacement is done since these functions are now closer to the third module than to the second module.

6.2 Matching process performance

In this group of experiments, we compare the contents of the recovered modules against the contents of the CLIPS files[11]. In our experiment, for each source file we defined two modules in AQL with the main-seeds selected from that file. Then the functions in the recovered modules were compared against the functions contained in the file.

[11] According to the CLIPS documentation, the CLIPS files have been designed as modules.

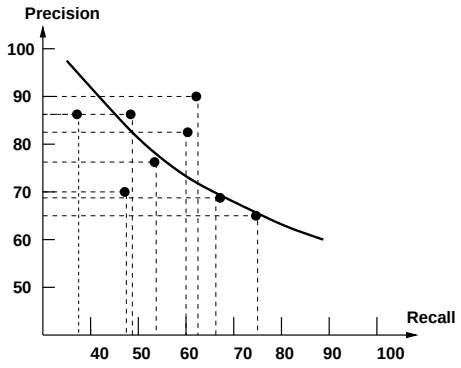

Figure 9. The interpolation curve for the obtained Recall and Precision values.

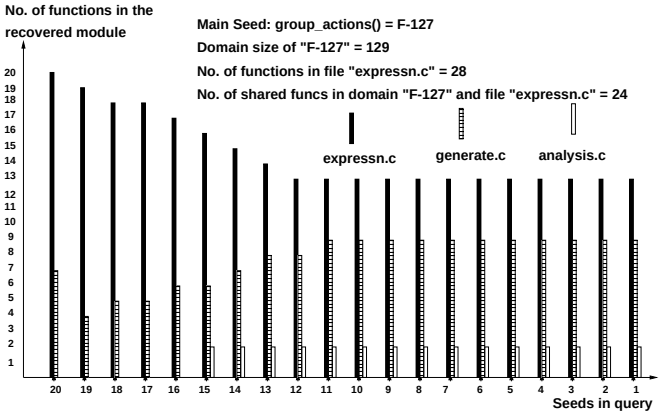

Figure 10. Investigating the stability of the recovery process with respect to the change of a module's seeds.

Figure 9 illustrates the interpolating curve between the resulting points in the *Precision-Recall* plane. The results indicate that our matching process behaves very well considering that we maintain a Precision level of 90% for a 60% level of Recall. Informally, this result means that we can retrieve 60% of the architecture as documented in the manual while having only 10% total noise in the obtained results.

Figure 10 represents the stability of the recovered modules with respect to the change of the selected seeds. For this experiment, 20 functions (seeds) including the main-seed (F-127) from the file `expressn.c` (consisting of 28 functions) were defined in a module to be recovered. The recovered functions in different files are shown using columns. A sequence of 20 experiments conducted. In each subsequent experiment we deleted one more seed (the least associated seed with the other seeds) from the query. The result shows that once the number of seeds in the query reaches to a threshold (here 12 seeds), the experiments recover the *same* group of entities until only the main-seed remains. Therefore, our recovery method tends to produce the same result when we increase or decrease the number of seeds considered in the query. This experiment has been performed for different files of the CLIPS system with similar results. From this experiment we conclude that we can break the recovery process of a module with a large search space into smaller steps and use the result of the initial steps as the seeds for the next steps.

7 Conclusion

In this paper we present a methodology for architectural design recovery based on data mining techniques. In this approach, a legacy system is parsed and its source code is represented as a variation of the data mining frequent itemsets representation. A structured query language is used to describe possible architectural design abstractions for the given legacy system. A pattern matching engine uses data mining, clustering, and a modification of the branch and bound search algorithm to obtain the components of the legacy system. The result is an architectural design with the best matching to the query specified by the user. Queries are incremental and allow to specify (describe) different parts of the system. A score is associated with each possible match which guides the pattern matching mechanism to rank architectural design alternatives and present an optimum design to the user for further evaluation. Initial results obtained by applying the proposed technique to medium size systems (30-50 KLOC) show that the technique is accurate and scalable. In particular, the technique has been applied on the architectural recovery of the APACHE web server, and BASH Unix shell. On-going work includes the evaluation of the recovery technique on larger software systems at the IBM Toronto Lab, Center for Advanced Studies.

References

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

[2] Architectural Design Recovery using Data Mining Techniques, Web site, URL = http://se.math.uwaterloo.ca/ksartipi/papers/tech-rep.ps.

[3] R. Agrawal, K. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[4] 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.

[5] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.

[6] Burnstein and K. Roberson. Automated chunking to support program comprehension. In *Proceedings of IWPC'97*, pages 40–49, Dearborn, Michigan, 1997.

[7] D. N. Chin and A. Quilici. Decode: A co-operative program understanding environment. *Software Maintenance: Research and Practice*, 8:3–33, 1996.

[8] C. M. de Oca and D. L. Carver. A visual representation model for software subsystem decomposition. In *WCRE: Working Conference on Reverse Engineering*, pages 231–240, Honolulu, Hawaii, October 1998.

[9] B. S. Everitt. *Cluster Analysis*. John Wiley, 1993.

[10] U. M. Fayyad. *Advances in knowledge discovery and data mining*. MIT Press, Menlo Park, Calif., 1996.

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

[12] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliche-based environment to support architectural reverse engineering. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 319–328, 1996.

[13] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In J. H. Johnson, editor, *Proceedings of CASCON'97*, pages 169–183, November 1997.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ 07458, 1991.

[15] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Recognizers for extracting architectural features from source code. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 252–261, Toronto, Canada, July 14-16 1995.

[16] R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE: Working Conference on Reverse Engineering*, Honolulu, Hawaii, October 1998.

[17] A. K. Jain. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, N.J., 1988.

[18] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42–50, November 1995.

[19] B. Lague, C. Leduc, A. L. Bon, E. Merlod, and M. Dagenais. An analysis framework for understanding layered software architectures. In *Proceedings of IWPC'98*, pages 37–44, Ischia, Italy, 1998.

[20] D. C. Luckham, J. J. Kenny, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[21] 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.

[22] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion model: Bridging the gap between source and higher-level models. In *In proceedings of the 3rd ACM SIGSOFT SFSE*, pages 18–28, October 1995.

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

[24] L. Rising and F. W. Calliss. Problems with determining package cohesion and coupling. *Software Practice and Experience*, 22(7):553–571, July 1992.

[25] M. Shaw, R. DeLine, et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[26] V. Tzerpos and R. C. Holt. The orphan adoption problem in architecture maintenance. In *Proceedings of the Working Conference on Reverse Engineering*, Amsterdam, October 1997.

[27] T. A. Wiggerts. Using clustering algorithms in legacy systems modularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE Computer Society Press, October 1997.

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