A Pattern Matching Framework for Software Architecture Recovery and Restructuring *

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 framework for software architecture recovery and restructuring. The user specifies a high level abstraction view of the system using a structured pattern language. A pattern matching engine provides an optimal match between the given pattern and a decomposition of the legacy system entities by satisfying the inter/intra-module constraints defined by the pattern.

The data mining technique Apriori is used by the matching engine to reveal meaningful data and control flow properties of the target system and limit the search space. A branch and bound search algorithm using a score function, models the constraints in the pattern as a Valued Constraint Satisfaction Problem (VCSP), and assists in searching for an optimal match between the given pattern and the target system.

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 [9]. Another approach is based on user defined constraints that need to be satisfied [18] and therefore, architectural recovery becomes a Constraint Satisfaction Problem (CSP). We propose an architectural design recovery based on design descriptions that are provided by the user in the form of queries using a pattern language. We call this pattern language, *Architectural Query Language* (AQL).

In the proposed approach the target system (consisting of a number of entities and relationships) is decomposed into a collection of modules with inter/intra-module constraints, defined using the AQL. The intra-module constraints require that the resulting architecture demonstrate high-cohesion among the module's constituents. The inter-module constraints, if manifested as import/export links, can control the coupling among the modules. In this sense, the AQL query provides a description of the *conceptual* architecture and the instantiated pattern (i.e. source code entities are assigned to the variables) provides the corresponding *concrete* architecture. The matching engine searches for an optimal arrangement of functions, types, and variables of the original system, into modules that conform with the user's view of the conceptual architecture.

We view the matching process as a Valued Constraint Satisfaction Problem (VCSP) with both implicit and explicit constraints at the conceptual architecture level. In the VCSP domain, the constraints can be violated and the goal is to find an optimal solution. Therefore, the valuation of the violated constraints are minimized in order to group similar values in a module and, at the same time, satisfy the constraints defined at the conceptual architecture abstraction.

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.

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

0-7695-0656-9/00 $10.00 © 2000 IEEE

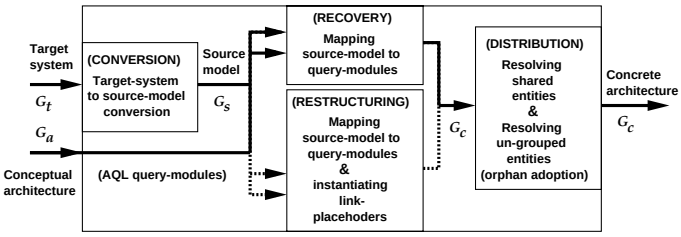


Figure 1. The proposed software architecture recovery framework

2 Related work

The following approaches are related to our work. The Murphy's reflexion model [13] 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 [12]. Specialized queries (recognizers) for extracting particular properties from the source code are presented in [6, 8]. In [3] a tool for code segmentation and clustering using dependency and data flow analysis is discussed. Holt [9] presents a system for manipulating the source code abstractions and entity-relationship diagrams using Tarski algebra. In [4] a clustering approach based on data mining techniques is presented. Lague et al. present a methodology for recovering the architecture of the layered systems [10]. 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 [16], Rapide [11], and ACME [7].

3 A framework for architectural recovery

Software architecture recovery has been extensively investigated in the literature. It has been mostly seen as a problem of identifying aggregate relations from the source code and applying clustering techniques to derive highly cohesive, low coupled modules.

We present the architectural recovery problem as a pattern matching problem. In particular, we define a pattern language to specify the architectural patterns as abstract queries. We call this pattern language Architectural Query Language (AQL). The proposed software architecture recovery framework consists of three phases (Figure 1):

In the first phase (*conversion*), the source code is represented as a typed, attributed, directed multi-graph G_t. Nodes in G_t represent source code entities (i.e., File, Function, Variable, Type). Edges in G_t represent data and control flow dependencies (i.e., calls, defines, sets, updates, declares). In the conversion process, the low-level relations between entities are aggregated into more abstract relations (i.e., calls and uses) which is suitable for architecture recovery. The result is the source model graph G_s. Similarly, the AQL query is represented as a multi-graph G_a where nodes correspond to high level design abstractions (i.e., Module, Subsystem) or, placeholders that are instantiated by the source code entities. Edges in G_a correspond to abstract design relations such as *imports*, *exports*, *contains* . These entities and relations, defined for each programming language, conform with a schema (*domain model*) for that programming language. So far, we have defined domain models for PL/I, PL/IX, PL/X, RPG, C, C++ and, Java.

In the second phase (*Recovery* or *Restructuring*), the pattern matching engine instantiates the placeholders in G_a with the source code entities in G_s. This instantiated graph G_a is denoted as the concrete architecture graph G_c. Ideally, we would like the graph G_c be isomorphic to a subgraph G_d, obtained from G_s. In this way, G_a and the pattern matching algorithm generate a decomposition G_d of G_s such that, G_c and G_d become isomorphic or similar. Since graph matching algorithms are computationally expensive, we formulate the matching process using a tree-based search space which is discussed in more detail in the following sections.

In the third phase (*distribution*), as a post-processing phase, the unresolved source code entities are bound to the non-instantiated placeholders. This phase addresses the *orphan adoption* problem [9].

This framework allows us to restructure an existing system by imposing *constraints* on the interactions between the modules. A common form of these constraints is defining import/export links among the modules in the AQL query. We define uninstantiated links between modules as constraints, imposed by the user, to be satisfied by the recovery process. The result of the recovery or restructuring process is a *concrete architecture* that conforms with the *conceptual architecture* as it is specified by the AQL query. We adopt a typed, attributed, directed graph formalism which is similar to some approaches in the literature [5]. The graph representation of the target system (G_t or, G_s) is defined as a tuple:

$$G_s = \langle \mathcal{V}_s, \mathcal{A}_s \rangle$$

where $\mathcal{V}_s$ is the set of typed vertices obtained from the target system (source code entities), and $\mathcal{A}_s$ is the set of allowable edges between vertices obtained from the language

domain model.

Similarly an AQL query can be represented as a graph and is specified as a tuple:

$$G_a = \langle \mathcal{V}_a, \mathcal{A}_a \rangle$$

where $\mathcal{V}_a$ is the set of typed vertices (placeholders), and $\mathcal{A}_a$ is the set of allowable edges (aggregate relations) between vertices.

The matching process instantiates the set of placeholders $\mathcal{V}_a$ with the entities from $\mathcal{V}_s$, hence, converting G_a to G_c.

During the matching process, the graph isomorphism problem can become intractable for real applications, therefore, we formulate our problem as a constraint satisfaction search problem where isomorphism is relaxed to graph similarity. In this framework, we use the data mining algorithm Apriori in a pre-process phase to reveal the bi-partite subgraphs of the target system graph G_s. The resulting source model graph G_s is used by the pattern matching process (as a constraint satisfaction search space) to identify the isomorphism between G_c and specific partitions of the graph G_s, as opposed to any arbitrary partition of G_s.

3.1 Target system representation

We use an Entity Relationship (ER) model to represent the source code artifacts in the form of a directed typed graph (i.e. G_t). Such a graph is represented using typed nodes that correspond to syntactic constructs of the software system under analysis obtained as a by-product of the parsing process, and edges that correspond to allowable relationships between the graph nodes. Allowable relationships between nodes are defined in terms of a domain model for a given programming language. For example if the implementation language is `C`, the domain model contains entities such as *File*, *Function*, *Identifier*, *Declaration*, and *If-Statement*. Allowable relationships in the `C` domain model include *Function calls Function*, *File includes Library*, *Function usesVar Identifier*.

3.2 Source model representation using data mining technique

The source model is an abstraction of the target system to a level that is suitable for architectural recovery. In this section, we discuss a source model representation that facilitates the process of mapping the entities from the source model onto the modules in query. We use the *Apriori* data mining technique [2] to discover a common pattern among a group of system entities that is not trivially observable.

Association strength values, obtained from the Apriori algorithm, are annotated to the nodes in the resulting source model graph G_s. These association values are used to partition the source model graph into highly-cohesive and low-coupled subgraphs as the result of the recovery process.

Most data mining algorithms are based on the concept of database *transactions* 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 2) that are called or used in any form by F_t.

Interesting properties of data in a database, namely *association rules*, are extracted from *frequent itemsets* [2]. 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 in Figure 2). 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.

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 (**V** for variable, **T** for type, **F** for function).

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*.

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 $Dom(s)$. Below, the set of entities in $Dom(s)$ (without the association values) are defined:

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

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 whole frequent itemsets F in the system are those 5 records that

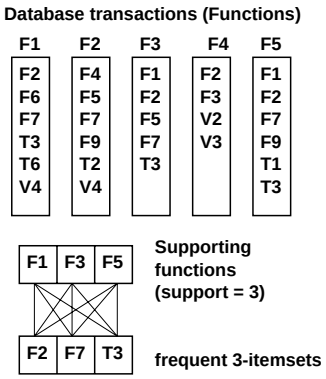


Figure 2. An application of the "database transaction" notion in Reverse Engineering domain.

we presented above, then the domain of function F-83 is as follows[1]:

Dom(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 frequent itemsets and their corresponding supporting functions demonstrate a high correlation among the group of supporting functions and the itemsets. Therefore, *k* (the cardinality of the itemset) can be viewed as the strength of the association among every pair of items in the itemsets and its supporting functions. A high association value among a group of system entitles qualifies them as candidates to be put in the same module.

This correlation can be demonstrated as complete bi-partite sub-graphs in the whole graph G_s of the source code entities. In this sense, the supporting functions and the itemsets are located at two opposite sides. Figure 3(a) is a typical representation of a graph that demonstrates the entities and relationships in a source model (*graph* G_s). The entities and relationships are an abstraction of those found in a software system. The generalized relationships include: *call*, *useVar*, and *useType*. Apparently, this graph lacks any interesting pattern to be used for guiding the recovery process. Applying the Apriori algorithm on this graph discovers the bi-partite sub-graph patterns, Figure 3(b).

The collection of domains (*Dom(s)*'s) is the basis for grouping the entities into modules, hence, this collection constitutes our source model representation. We obtain *n* domains (*n* is the number of distinct entities in the frequent itemsets), whereas, we need only *m* ($m << n$) domains, one

[1]In this example, we use the number of supporting functions as the association value.

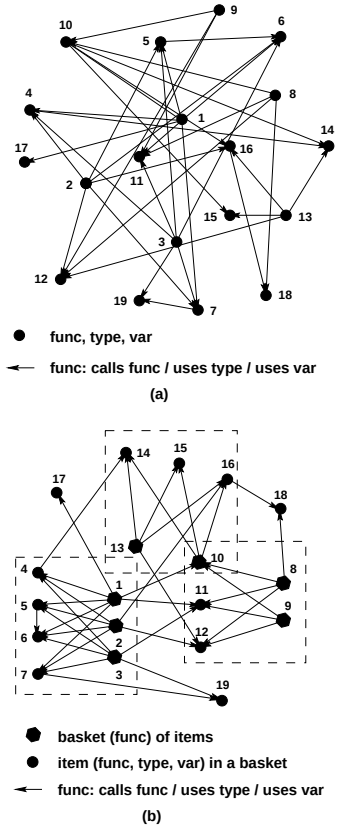


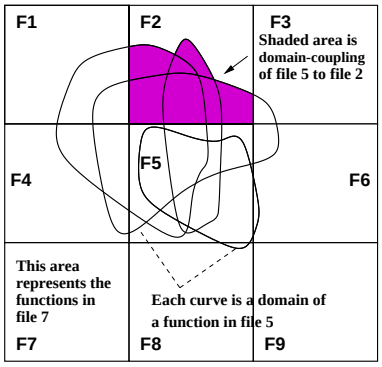
Figure 3. (a) An un-processed graph representation of the target system. (b) A bi-partite sub-graph representation resulted from applying the Apriori algorithm on graph (a).

for each module to be recovered. In a pre-process phase, a domain selection algorithm assists us in selecting the best *m* domains. The selection process is based on the size of the domain and the level of the association strength between its contents.

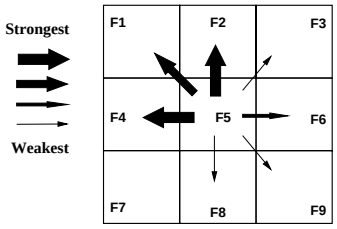
This process provides a means for partitioning the search space, hence, reduces the complexity of the matching process.

3.3 System partitioning based on domain coupling

A system can be partitioned into subsystems based on the notion of *domain coupling and cohesion* as a generalization of the conventional coupling between the system components (e.g., file, module, subsystem). Conventionally, the coupling between two system components is defined as the existing dependency between the components based on a single criterion (e.g., calling dependency among



(a) Domain-coupling among files



(b) Demonstration of the strenght of the domain-couplings in part(a).

Figure 4. Demonstration of the notion of domain-coupling and domain-cohesion among the functions of the files in a system.

the functions). The domain-coupling is defined as an overall dependency between two system components based on an overall association between these components with regard to several criteria. In Figure 4(a), a system of nine files is shown. The domain-coupling of file 5 onto file 2 is proportionally related to the size and density of the shaded area to the area of the file 2 (i.e., the functions of file 2). The domain-cohesion is defined for a single file as the proportion of the overlapped part of the file's functions domains to the same file's functions. Therefore, a file whose domains of functions are concentrated on the functions of the file itself, is highly cohesive. Figure 4(b) demonstrates the domain-couplings of the file 5 onto other files of the system. In this example, file 5 has high coupling to files 2 and 4, medium coupling to file 1, and low-coupling to other files, therefore, file 5 is not cohesive. The thickness of the arrows in Figure 4 can be shown by different colors between the files of a system in a visualization tool such as Rigi. This method allows us to group the files into subsystems according to the strength of couplings to each other.

4 An abstract 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 is used for: i) decomposing the program representation into modules with inter-/intra-module relationships; and ii) 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 numget()
    IMPORTS:
      FUNCTIONS:    func $IF,
                    func ?F1, func ?F2, func ?F3
      TYPES:        type $IT
      VARIABLES:    var $IV
    EXPORTS:
      FUNCTIONS:    func $EF,
                    func ?F4, func ?F5
      TYPES:        type $ET
      VARIABLES:    var $EV
    CONTAINS:
      FUNCTIONS:    func $CF(4..20), func numget()
      TYPES:        type $CT(1..3)
      VARIABLES:    var $CV(0..2)
END-ENTITY

MODULE: M2
    MAIN-SEED:      func generic_compute()
    IMPORTS:
      FUNCTIONS:    func $IF,
                    func ?F4, func ?F5
      TYPES:        type $IT
      VARIABLES:    var $IV
    EXPORTS:
      FUNCTIONS:    func $EF,
                    func ?F1, func ?F2, func ?F3
      TYPES:        type $ET
      VARIABLES:    var $EV
    CONTAINS:
      FUNCTIONS:    func $CF(4..20),
                    func generic_compute()
      TYPES:        type $CT(0..2)
      VARIABLES:    var $CV(0..2)
END AQL
```

The prefixes "$" and "?" represent simple-placeholders and matching-placeholders, respectively. For example $CF(4..20) denotes simple-placeholders that can be instantiated by *minimum* 4 and *maximum 20* functions that are

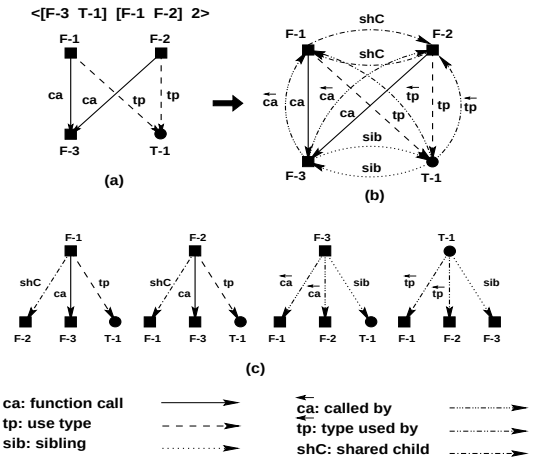


Figure 5. Transformation steps from frequent itemset to n-tree graph.

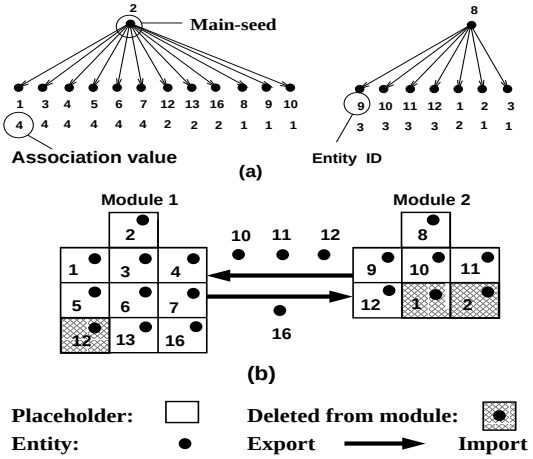


Figure 6. (a) Tree representation of two main-seed domains in Figure 3. (b) Modeling the recovery process in a simplified example.

contained in a module [2], and $IF denotes an unidentified number of simple-placeholders that is determined by the *distribution* phase in Figure 1.

Two matching-placeholders, with the same name in different parts of a query, can only be instantiated with a single entity, and represent the links between those two modules. The matching process provides an instantiations which bind these AQL placeholders with actual entities of the source model. When *all* placeholders in the query have been instantiated, i.e., bound to values (even by a *nil* binding), a concrete system architecture is generated (as opposed to the abstract architecture defined by the AQL query).

5 Modeling the recovery process

We perform a transformation onto the frequent itemset representation of the domains (section 3.2) in order to demonstrate: i) a modeling of the recovery process, and ii) a clear representation of the domains suitable for domain selection algorithm (explained below). This transformation converts the frequent itemsets onto a forest of *n* trees (Figure 5). In this conversion, each node of the connected graph is allowed to appear in several trees.

The transformation steps are explained with reference to the parts of Figure 5 as follows:

(i) A tuple representation and a bi-partite graph representation of a frequent 2-itemset [F-3 T-1] with support 2, i.e., [F-1 F-2], are considered.

(ii) The complete bi-partite graph in Figure 5(a) is transformed into a complete graph, in two steps: 1) two relations *sibling* and *sharedChild* are added to the graph, 2) each relation between a pair of nodes is coupled with its inverse relation (Figure 5(b)).

(iii) Each node in Figure 5(b) along with its connected *n - 1* nodes (*n* is number of all nodes) constitute a tree. The result is a forest with *n* tree components (Figure 5(c)).

Each tree represents a domain of a candidate main-seed to be considered for a module. The domain selection algorithm then performs an exhaustive search to find the best candidate domains for the modules in the query. The criteria for this search include: i) high average level of associations between each entity in a domain and the corresponding main-seed; ii) low level of scattering of the domain entities into the system files; and iii) large domain size.

Figure 6(a) illustrates two domains corresponding to two main-seeds #2 and #8 (see Figure 3(b)), selected by the domain selection algorithm. Figure 6(b) illustrates a highly simplified mapping[3] from the selected domains onto the modules. In this model, the matching process selects the entities for each module solely based on the highest value of the data mining association (i.e, from left to right of the domain trees, Figure 6(a)). Each shaded box (i.e., entities #1, #2, and #12 in Figure 6(b)) denotes that the corresponding entity is closer to the other module, therefore, it is deleted from the current module. The import/export link handling is as follows: all entities that are called or used by a particular module, and exist in other modules, are imported by the

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

[3] In this model we only consider the data mining association values as the criterion for closeness.

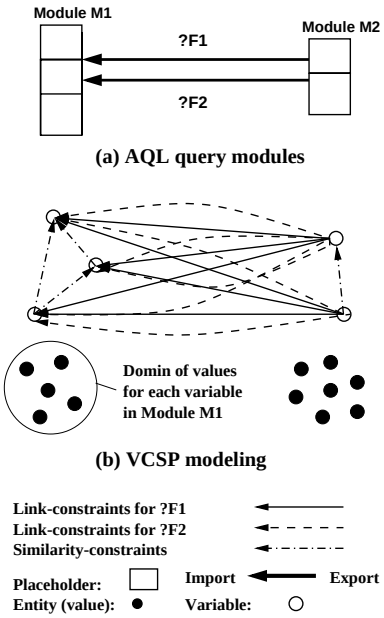


Figure 7. Mapping from modules in query (a) into the VCSP domain (b).

former module and are exported by the latter modules. In Figure 6(b), The imported/exported entities are also shown (i.e. entities 10, 11, 12, 16).

As we noted in section 3, the goal of architectural recovery (or restructuring) is to satisfy the inter/intra-module constraints that are defined in an abstract form (e.g., using a query language notation). One manifestation of these constraints includes: i) instantiating the modules' contained placeholders to provide cohesive modules; ii) instantiating the link placeholders such that they carry the imported/exported entities. As there are many possible ways of instantiating links between modules, the whole problem can be reduced into a Valued Constraint Satisfaction Problem (VCSP) by translating the above abstract constraints into exact constraints between variables.

Therefore, we first define an abstraction of the recovery problem using the main-seeds (one for each module) and the placeholders in different parts of a module in query. Then we translate these abstract constraints into exact constraints between variables to be solved by a known problem solving domain, i.e., *Valued Constraint Satisfaction Problem* (*VCSP*), to be dealt with.

5.1 Valued Constraint Satisfaction Problem

The *Valued Constraint Satisfaction Problem* framework (*VCSP*) [15] is an extension of the *Constraint Satisfaction Problem* framework (*CSP*), that allows *over-constraint* problems to be dealt with. In the VCSP framework, a *valuation* is associated with each constraint. The valuation of an assignment is defined as the aggregation of the valuations of the constraints which are violated by this assignment. The goal is to find a complete assignment of minimum valuation. A VCSP framework is defined as a quintuplet $P = (V, D, C, S, \phi)$, where V is a set of variables, D a set of associated domains, C a set of constraints between the variables, S a valuation structure, and ϕ a valuation function.

For this work we use a valuation structure which is known as Σ-*VCSP* (additive VCSP) and is discussed in [15]. Moreover, we use the *branch and bound* search algorithm [14] to find a minimum valuation of the assignment of the entities in the source model (domain of the variables) into the placeholders (variables) of the query.

In addition to our previous work [14], in this paper we: i) improved the search engine to handle VCSP; ii) provided automatic main-seed selection mechanism; iii) used the notion of domain-coupling as a means for decomposing the system into subsystems to be further used for modularization and; iv) provided interfaces to web browsers and graph visualizer RIGI. In the next section the modeling of the architectural recovery with Σ-*VCSP* is discussed.

5.2 Modeling architectural recovery using VCSP

In this section, we describe the mapping between a simple AQL query of two modules *M1* and *M2* (see Section 4), and its associated VCSP model (Figure 7). The steps of this mapping are as follows:

Step 1
For every node in each module, we assign a variable v_i in the set of variables *V*, and assign a corresponding domain d_i in the set of domains *D*, where d_i = *Dom*(s) (i.e., the domain of the main-seed s in the corresponding module).

Step 2
For every pair of variables in *V* that correspond to the same module (e.g., $M1$), define a constraint of type *similarity-constraint* in *C*. If module *M2* exports a *matching-placeholder* link to module *M1* (e.g., ?F1), assign a constraint of type *link-constraint* from every single variable in module *M2* to every single variable in module *M1* with the same name as the matching-placeholder. Repeat the same procedure for every matching-placeholder link between each pair of modules in the query (Figure 7) .

We define the valuation function ϕ on the basis of our architectural recovery objectives as follows: i) the average similarity value between the group of entities in a module must exceed a threshold which is determined by the overall properties of the software system; ii) all import/export links between the modules must be instantiated.

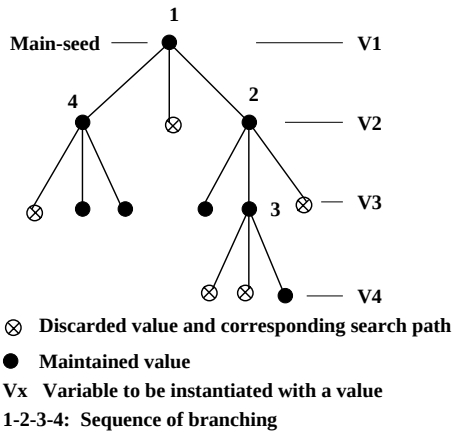


Figure 8. The branch and bound search tree.

In order to meet the above general requirements, we define the condition for satisfaction or violation of each type of constraints between a pair of variables in V we defined earlier:

- *similarity-constraints*: the similarity between each pair of the assigned variables (entities) in a module is determined by considering the shared features of those two entities (measured using the Jaccard formula[4], where A and B represent the sets of feature values for two entities [17]) as well as the highest association value between the corresponding entities. We assign a very high similarity value for satisfaction of a similarity-constraint so that almost all such constraints are violated. This causes the valuation function ϕ to aggregate the distance values (1 - similarity value) between the candidate entity and the entities of the already instantiated placeholders in that module, as a measure of ranking the module by the branch and bound algorithm (Figure 8). If the aggregated cost of the violated constraints exceeds an upper-bound, the candidate value is discarded and the search tree for that value is pruned. If such a incidence is repeated for all domain values of a variable, a form of backtracking occurs.

- *link-constraints*: if a link-constraint is satisfied (e.g., the relation *call* or *use* exists between the values of the corresponding variables), all other link-constraints with the same name (e.g., $?F1$) are deleted from C. If such a constraint is violated[5] and the number of the uninstantiated variables in the current module is less than the uninstantiated links between the corresponding modules, the violation cost is maximum, therefore, the candidate value is discarded. Otherwise, the violation cost would be very small and is added to the cost of violation of the similarity-constraints for this value.

With the above valuation strategy, the steps for the branch and bound search algorithm are as follows:

Step 1
the next variable is selected from the current module to be instantiated;

Step 2
from the domain of this variable the next value (candidate value) is selected to be assigned to the variable;

Step 3
all similarity-constraints and link-constraints between the assigned variables are evaluated and checked for satisfaction/violation;

Step 4
if the cost of violation is very high (i.e., the upper-bound), the candidate value is discarded, else, the valuated cost is used as the ranking criterion for the current module and the current module is put in the proper place of the list of all partially assigned modules for future assignment and ranking (more detailed discussion in [14]). Figure 8 illustrates the behavior of the employed branch and bound algorithm in order to come up with an optimum solution.

6 Experiments

In this section, experimental results obtained using the proposed system are presented. Our experimentation platform consists of a Sun Ultra 10 (333MHZ, 256M memory). It takes 4 minutes to parse the target system (CLIPS)[6], 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 user is a part of the recovery process and modifies and enhances the query based on the result of the previous run. A typical scenario for architectural recovery with this tool proceeds with the following steps.

- The user parses the target system using the Refine's parser, generates a database from the system entities and their relationships, and uses the Apriori algorithm to produce the source model representation of the system.

- The user decides on the part of the system for recovery (a sub-system or the whole system) and defines the number of the modules to be recovered. The

[4] Jaccard = $\frac{|A \cap B|}{|A \cup B|}$

[5] Handling this case is different for the *import* constraints. The current discussion is for *export* constraints.

[6] An expert system with size 40 KLOC.

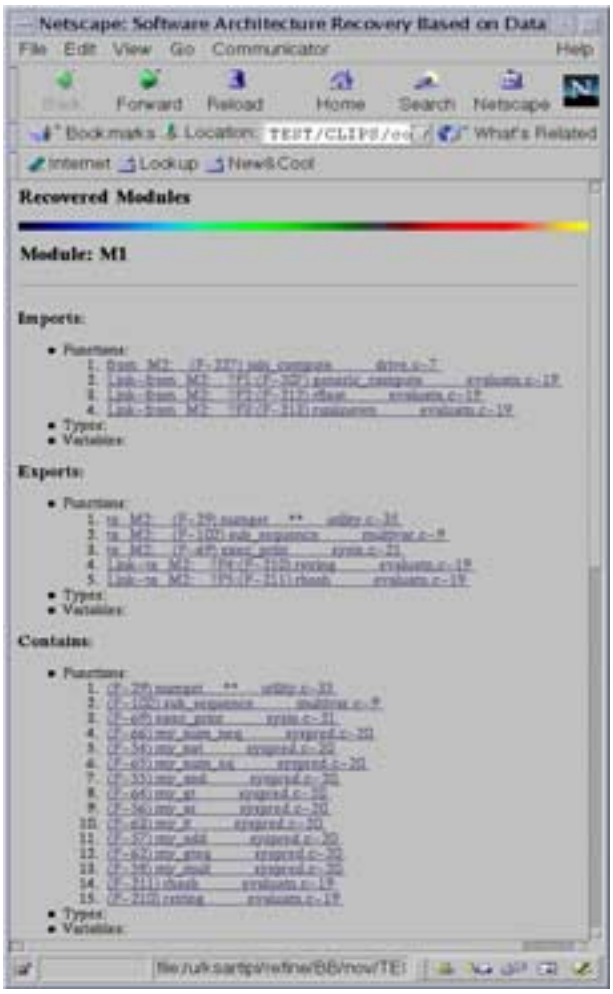


Figure 9. The result of the recovery is presented using the Netscape browser.

domain selection algorithm then searches the source model based on the user preferences (discussed in section 5) and produces the template of the query. The user can run the template query directly, or first tailor it to his/her needs and then run it. Different modes of operation such as automatic and incremental in terms of the number of modules, or the type of the entities (i.e., function, type, and variable to be recovered), provide a convenient environment for the recovery process.

- The user observes the result of recovery through Web browsers with hypertext links to the actual entities in the source files, and investigates the property of the recovered modules through Rigi visualization tool [1].

- Based on the result of recovery, the user decides on restructuring the recovered module based on ill-formed modules, uneven module size, or unbalanced inter-module interactions. The restructuring is performed by defining import/export links among the modules, as hard constraints to be satisfied, and by changing the size of the modules. The tool then provides new restructured modules in which the links are forced to be instantiated.

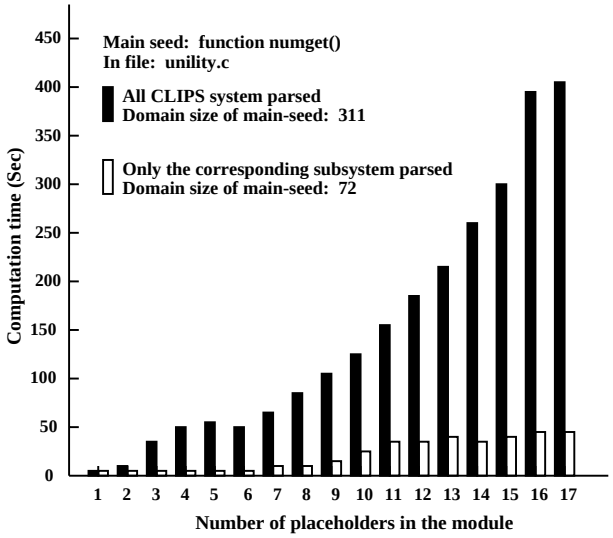


Figure 10. The run time diagram for module recovery.

6.1 Module recovery

In this section, the result of the recovery process corresponding to the query of section 4 is discussed. Figure 9 shows the recovered module M1 using the Netscape web browser. Each line corresponds to an entity (here only functions) with its name, its container file name, and the linked modules (for import/exports). The module's main-seed has been labeled using "**". Each entity has a hypertext link to its corresponding actual text in the system's source file. Three matching-placeholders from M2 to M1, and two matching-placeholders from M1 to M2 have been instantiated in the recovery process. These links were defined as the constraints between these modules. The four links F-227, F-29, F-102, and F-69 have been instantiated in the distribution phase of the process (Figure 1). The file `utility.c` provides services to the other files of the system. In this recovery, the dependency of the files `sysio.c, evaluatn.c` and `multivar.c` to this file is shown.

Figure 10 illustrates the results related to the time com-

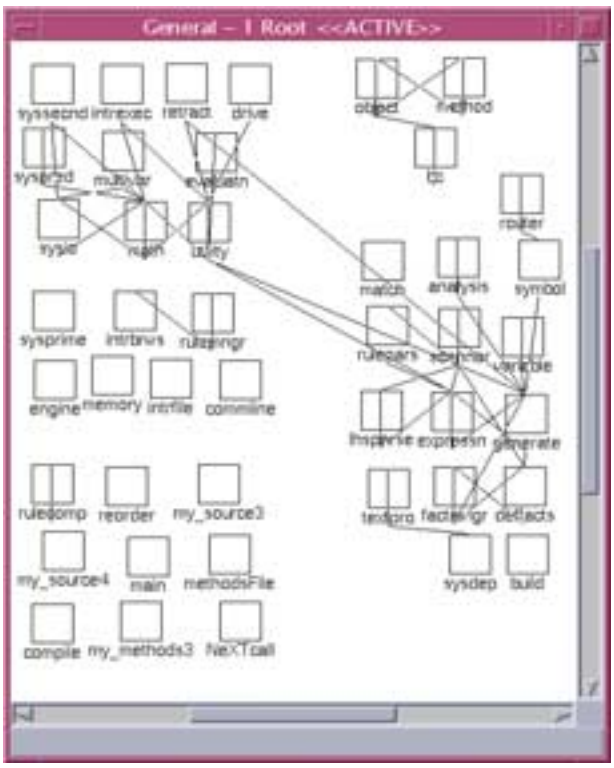


Figure 11. The decomposition of the CLIPS system files into subsystems.

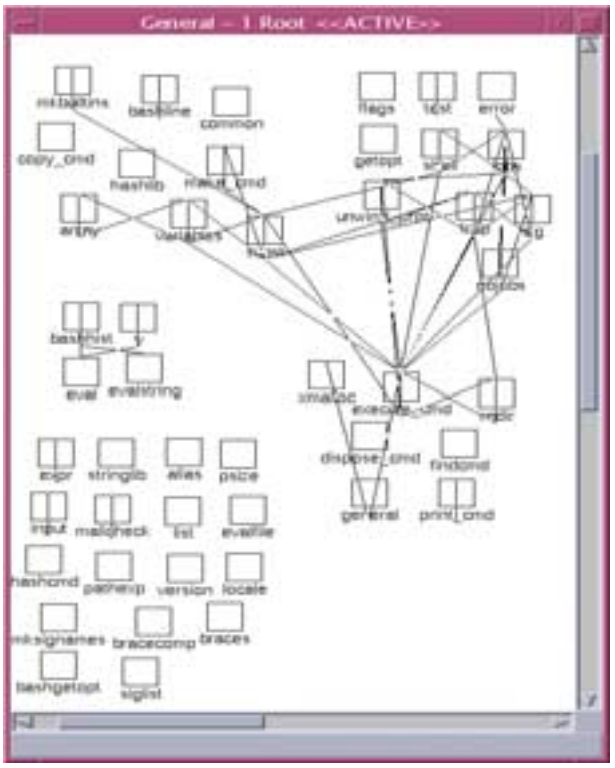


Figure 12. The decomposition of the BASH system files into subsystems.

plexity of the recovery of the module M1. In this experiment, we started from the result of the recovery (discussed above) and put 18 functions (seeds) and one placeholder in the query for the module to be recovered. We run the experiment and registered the time. At each subsequent experiment we deleted one more seed from the query, considered it as a placeholder, and then run the matching process again. The result shows that the time increases rapidly by the increase of the number of placeholders in the module.

Experiments on matching process accuracy using precision / recall evaluation and matching process stability through the result of recovery can be found in [14].

6.2 System decomposition

We decompose the CLIPS and BASH systems (Figure 11 and Figure 12 respectively) using the system partitioning discussed in section 3.3. In these Figures only the strong and medium couplings are shown using the lines between the boxes (the colors are not distinguishable). A line from the bottom of a box to the top of another box signifies that the former is coupled on the latter in terms of using the services defined in the latter file. Figure 11 shows the decomposition for the CLIPS system into five subsystems. We compared this result with the CLIPS documents. The subsystems at the bottom-right corner of Figure (with 15 files) corresponds to the subsystem *Parsing Modules* in CLIPS. Data flow analysis and cohesion metrics indicated that the files in this subsystem are highly coupled to each other. The subsystem at the top-left corner of the Figure (with 10 files) corresponds to the subsystem *Inference Engine Modules*, and the subsystem at the top-right corner (with 3 files) corresponds to the subsystem *Object*. Figure 12 illustrates the same experiment with the Bash system (Unix shell).

7 Conclusion

In this paper we presented a framework for software architecture recovery. We adopt a directed, typed, attributed graph formalism to provide a unified environment for the framework artifacts to be defined and related. A structured query language is used to describe architectural design abstractions for the given software system in the form of modules and high-level constraints. The software system is parsed and a data base of the system entities and their relationships is generated. The data mining technique Apriori

is used to provide a restricted and highly associated source model for the recovery process. The translation of the high-level constraints into exact constraints reduces the architectural recovery into an over-constraint system of variables and constraints to be dealt with in the Valued Constraint Satisfaction Problem (VCSP) domain. Initial results obtained by applying the proposed technique to medium size systems (30-50 KLOC) are promising that the technique is reasonably accurate and scalable. 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] 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.

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

[4] 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.

[5] T. R. Dean and J. R. Cordy. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313, April 1995.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architecture design recovery using data mining techniques. In *4th European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pages 129–139. IEEE, February 29 - March 3 2000.

[15] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of IJCAI-95*, pages 631–637, 1995.

[16] 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.

[17] 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.

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