An Amalgamated Dynamic and Static Architecture Reconstruction Framework to Control Component Interactions

Kamran Sartipi and Nima Dezhkam
Dept. Computing and Software, McMaster University
Hamilton, ON, L8S 4K1, Canada
{*sartipi, dezhkan*}*@mcmaster.ca*

Abstract

View-based software development is well adopted in forward engineering. However, most reverse engineering techniques still consider a single view of a software system with restricted scope of analysis. In this paper, we propose a novel approach that amalgamates dynamic and static views of a software system. The dynamic view is represented through profiling information that is extracted from executing a set of task scenarios that cover frequently used software features. The obtained profiling information is then embedded into a static view recovery process. We propose a pattern based structure recovery, as static view, that defines the high-level architecture of the software system using abstract components and interconnections that is defined using an architecture query language (AQL). In this context, both static and dynamic aspects of the software system are used to collect software entities into cohesive components whose dynamic interactions can be controlled. The whole recovery process is modeled as a Valued Constraint Satisfaction Problem (VCSP). A case study with promising results on the Xfig drawing tool has also been presented.

KEYWORDS: Multi-view recovery; Valued Constraint Satisfaction Problem; Data mining; Profiling; Pattern matching; Software architecture recovery; Scenario.

1. Introduction

Modern industrial organizations in different application domains are deeply involved in various software engineering tasks such as: developing new systems, integrating legacy assets with modern applications, decentralizing monolithic systems, and performing various maintenance activities. In order to sustain their competitiveness in industry, these organizations require a well maintained high quality software system to continue their business without any interruption caused by software failure.

On the other hand, software solution providers seek their customers among such organizations that lack enough in-house software expertise to maintain the quality of software system during the evolution of their hybrid systems. In both cases, a multi-view and interactive assistant-tool would be extremely valuable in order to identify the intended software system components, and to leverage the knowledge of the software experts about the impact of the integrated features on the system structure.

One key aspect that warrants the success of such solution providers is the richness of the collection of reverse engineering tools in possession to be able to provide relevant views that directly address customer needs which are changing over time. Once a problem is identified and solved, there will be the next issue which might require a different view.

To pursue this goal, in recent years we have developed a set of reverse engineering techniques and tools to extract information from: i) *static view*: clustering, pattern-based, and mined-association-based component recovery; ii) *combined static and dynamic views*: embedding profiling information into pattern-based component recovery, mining software features in source code, design pattern extraction using dynamic traces; and iii) *evaluation*: association and edge based architectural evaluation techniques. These techniques have been developed within "*Alborz multi-view and wizard-based toolkit*". Also, to enhance the usability by industry and research community, these tools are being migrated, or have been implemented, as Eclipse plug-ins.

In this paper, we present a scenario guided dynamic and static analysis with the goal of controlling the dynamic interactions among cohesive components in a pattern based structure recovery process. In this approach, the dynamic (or behavior) view is represented as the realization of frequently used task scenarios by the source code functions. We obtain profiling information in terms of the number of function invocations for each function that is involved in execution of frequent task scenarios. The obtained dynamic information is then embedded into the extracted source graph of the software system to be used for an amalgamated

static and dynamic view recovery process. The static (or structure) view is generated using an approximate pattern matching technique that is modeled as a Valued Constraint Satisfaction Problem (VCSP) [15]. In this context, a high-level architecture of the software system is defined using abstract components and connectors. Where each abstract component consists of a group of placeholders (as VCSP variables) to be instantiated by the system functions (as VCSP values) and an abstract connector is a collection of function invocations whose cardinality and invocation frequencies (as the VCSP constraints) can be controlled by the defined pattern. The architectural pattern is defined using our proprietary language, namely Architecture Query Language (AQL) which has been inspired by the ADL's design. The search engine performs an approximate matching operation that assigns values (functions) to variables (placeholders) that best satisfy the static and dynamic constraints.

In order to evaluate the proposed approach, using a case study of an interactive drawing application a pure-static approach is compared with our proposed joint static/dynamic approach. In general, a poorly maintained system through adding ill-designed patches for different system maintenance activities, causes feature scattering anomaly among the components which introduces a high volume of dynamic links between the system components. The proposed approach is a means for restructuring a legacy software system to improve both dynamic and static interactions among components.

The contributions of this paper are as follows: i) providing a pattern-based structural reconstruction technique to control the dynamic interaction of cohesive components; ii) modeling a multi-view reconstruction approach as a Valued Constraint Satisfaction Problem; and iii) enhancing *Alborz* toolkit [14] to perform the proposed multi-view recovery.

The rest of this paper has been organized as follows: Section 2 discusses the related research work from the literature. Section 3 presents an overview of the proposed multi-view approach. In Section 4 we elaborate on the steps taken to produce dynamic information. Section 5 is allocated to our interactive pattern matching process and modeling the recovery process. Section 6 presents the approximate pattern matching process. Section 7 discusses our experimentation with the proposed technique. Finally, Section 8 summarizes and concludes our discussion.

2. Related work

The proposed research in this paper is related to the approaches in software architecture view recovery that extract more than one view of the software system.

In a previous work we proposed an orchestrated multi-view software architecture reconstruction environment, where design, behavior, and structure views of a software system are extracted in a sequence, i.e., each extracted view is used as the seed to generate the next view [11]. The method utilizes feature-specific task scenarios to localize software features in the source code by obtaining frequent patterns in the scenario execution traces. The obtained core functionalities of the software features are used as the seeds in a software clustering process, hence incorporating behavior semantics into structure recovery. In contrast, in this paper, we use profiling techniques to extract behavior view; and structure view is generated using pattern matching techniques. Moreover, instead of sequentially connecting, the recovered views in our approach are explicitly merged together.

Vasconcelos et al. [17] present a dynamic reverse engineering approach that extracts the process and scenario views (from 4+1 views) of Java applications in the form of UML sequence diagram and use-case scenarios that are further combined with a static view using a toolkit called Odyssey [3]. Similar to our approach, they use dynamic analysis to recover the behavior view of the system along with complementary views. Riva et al. [9] propose a technique for architecture recovery using combined static and dynamic information. Their technique is based on choosing a conceptual architecture and also applying abstraction techniques on source code to manipulate the conceptual architecture. Their technique allows for the creation of domain-related architectural views for the architecture description of the system. Similarly in our approach, we use scenarios with design-derived features to guide the multi-view recovery process. In a similar context, Deursen et al. [16] propose a view-based software reconstruction framework that provides a common framework for reporting reconstruction experiences and comparing reconstruction approaches. Richner et al. [8] propose an approach to extract static and dynamic views from Java programs. Similar to our approach, they form a connection for information exchange between the two views.

3. Amalgamated static and dynamic model

Figure 1 illustrates the proposed interactive and pattern-based environment for software analysis using both static and dynamic properties of software towards a component based architecture recovery. In this regard, we enhanced the Alborz [10] architecture recovery toolkit to accommodate dynamic information in combination with static information and utilize a pattern matching technique that is modeled as a Valued Constraint Satisfaction Problem (VCSP). The architecture recovery process has three major stages, as follows.

Stage 1 (Static pre-processing). A comprehensive discussion of the static pre-processing stage has been

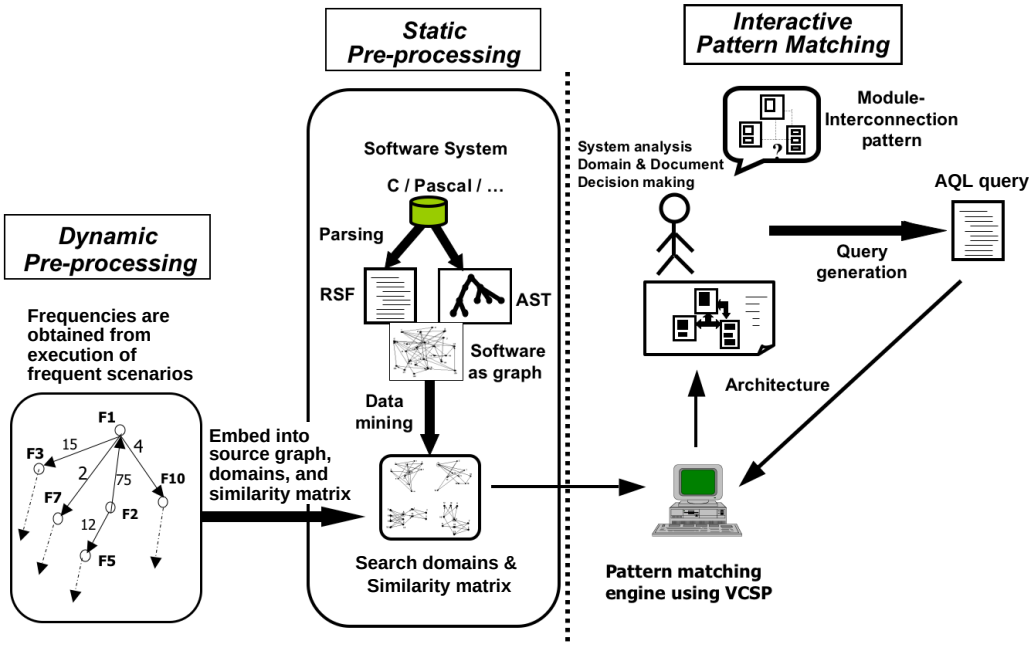

Figure 1. Multi-view and pattern-based Alborz architecture recovery environment. The profiling information generated by frequently used task scenarios are embedded into the static view to be used for controlling the interaction traffic between extracted components.

presented in [13]. In this stage, the software system is parsed to generate an *abstract syntax tree* (AST) and then using a schema, namely *abstract domain model*, it is transformed into a graph representation at a higher level of abstraction called *source graph*. Such a domain model provides programming language independence for the recovery process. The source graph is represented using a typed attributed relational graph notion defined in [6]. In this notation, nodes represent software constructs, such as *functions*, *data types*, and *global variables*, and edges represent relationships between these constructs, such as *function-call*, *datatype-use*, and *variable-use*. In this stage, an association-based data mining algorithm [4] is applied on the source graph that allows us to decomposes the source graph into smaller regions (search domains) where each search domain consists of a number of entities that are associated with a distinguished entity in that domain, namely *main-seed*. Finally, in the static pre-processing stage an association-based similarity matrix is generated that contains the mutual *similarity* values of every pair of entities in the system. The generated matrix will be used by the pattern matching engine in order to extract highly cohesive components.

Stage 2 (Dynamic pre-processing). In the dynamic pre-processing stage, first the software system is instrumented using a dynamic profiler tool such as *gprof* [2]. Then, a set of task scenarios that cover common features of the system are executed, and the list of functions and the number of their invocations (i.e., call frequencies) are captured and embedded into source graph and consequently into both the search domains and the similarity matrix of the static pre-processing stage. This stage is discussed in more detail in Section 4.

Stage 3 (Interactive pattern matching). In this stage, the user defines a conceptual architectural pattern (or architectural pattern for short) as a collection of abstract components and abstract interconnections, where each abstract component represents a group of placeholders to be instantiated by the system functions, and each abstract connector between two components represents both static and dynamic interactions between two components that can be defined as the architectural constraints. This architectural pattern is defined using a main-seed selection mechanism that searches the whole search domains and generates a ranking list of *top N* search domains with their detailed specifications. The user then selects the best search domain from the list to be used for the next component's recovery pro-

cess. Finally, the VCSP based search engine will search to find approximate matches between the defined architecture and the functions obtained in the current search domain while satisfying constraints with regard to the cardinality of the connectors and their dynamic interactions. This stage is discussed in more detail in Sections 5 and 6. Below, the relevant terminology used in this paper are defined.

Terminology

- *Feature:* a feature is a realized functional or non-functional requirement. However, in this paper only functional features are relevant.
- *Frequent feature:* a frequent feature is used by the users of the system more often than other features. Identifying frequent features requires domain knowledge and statistical study on the usage of the system.
- *Scenario:* a scenario is a sequence of features that trigger actions of the system and yield an observable result to the user [5].
- *Instrumentation:* refers to the process of inserting particular pieces of code into the software system (source code or binary image) to generate a profile of the software execution.
- *Component:* a component is a group of functions that implement a computational unit of a system. Components have import and export relations with other components.

4. Dynamic pre-processing

In the dynamic pre-processing stage, a set of common task scenarios are executed on the system and the resulting execution profiles (in terms of function call frequencies from different calling functions to each called function) are captured. This involves the following two steps: i) instrumentation of the software system; and ii) execution of a set of scenarios on the instrumented system and analysis of the resulting execution profiles. In the rest of this section the above two steps are discussed.

Software system instrumentation
We use the GNU profiler, gprof, to instrument the software system and capture the execution profiles. In order to use this tool, one has to compile the source code of the program using the *gcc* compiler with profiling options enabled. This tool provides two types of output: *flat profile*, and *call graph*. The flat profile provides the total amount of time that the program spends for execution of each function, whereas the call graph provides the amount of time that is spent in each function and its children. It also provides the number of times each function calls its children and is called by its parents. In this research, we use the call graph output of the profiler.

Scenarios execution and analysis of profiles
The familiarity of the user with the system's functionality and its operation assists in designing the scenarios, however this is not necessary. The scenarios should cover a set of "frequently used features" of the system, obtained from user's manual, application domain, and documents such as activity diagrams. To obtain valid function invocation frequencies in the profiles, the user should eliminate redundancies from the set of scenarios. This means a feature should not be executed more than its normal usage that is determined by domain knowledge. Finally, the set of selected scenarios are executed on the instrumented system. We extract the highest invocation frequency for each function-call from the execution profiles corresponding to different task scenarios. In a further step, the obtained frequencies are embedded into the source graph as the *frequency attribute* of the edges. This attribute will be used in the cost calculations of the Valued Constraint Satisfaction Problem (VCSP) model of the pattern matching, discussed in the following sections.

5. Interactive pattern matching process

We perform an interactive pattern matching process that is modeled as a Valued Constraint Satisfaction Problem (VCSP). The pattern matching process incrementally generates concrete software components that approximately match with the provided architectural pattern using our proprietary AQL language. In this context the VCSP modeling is sensible since the architectural pattern consists of soft constraints (i.e., highly similar functions within each component) and hard constrains (i.e., strict limits on the number of static and dynamic function calls among components) that can naturally be modeled by VCSP framework. The interactive pattern matching consists of two major parts: i) defining a query to represent the architectural pattern, and ii) performing the approximate pattern matching to obtain concrete components that conform with the constraints defined in the architectural pattern.

Modeling architectural pattern using AQL
We present an overview of our enhanced Architectural Query Language (AQL) that is used for describing a pattern of abstract components (or modules) and their constrained connectors (interactions among the components) that will be used by a matching process to identify a close match (as concrete architecture) within the structure of the subject legacy system. The syntax of AQL encourages a structured

description of the architecture. A typical AQL query is illustrated below:

```
BEGIN-AQL
MODULE: M1
    MAIN-SEEDS:       func  canvas_selected
    IMPORTS:
      FUNCTIONS:      func  ?IF,
                      func  ?F1(0 .. 4) M2
                      func  ?F2(180 .. 1) M4
    EXPORTS:
      FUNCTIONS:      func  ?EF
    CONTAINS:
      FUNCTIONS:      func  $CF(4 .. 30),
                       func  canvas_selected
    RELOCATES:        NO:
END-COMPONENT
    .....
END-AQL
```

In the above query the syntax for Imports (Exports) Functions "?Fu(x..y)" represents an unidentified group of links (i.e., function invocations) with the group number "u", where "x" represents the *maximum dynamic interaction* (i.e., maximum of function invocation frequency) and "y" represents the *maximum static interaction* (i.e., maximum static link quantity) between the corresponding components (modules). The above AQL fragment is interpreted as: module M1 with the main-seed function *canvas_selected* has 30 variables (function placeholders) that can be assigned by a maximum of 30 functions (represented by "$CF(4 .. 30)"). Module M1 may import up to four functions from module M2, however no dynamic interaction exists under the executed scenarios (shown as "?F1(0 .. 4) M2"). Module M1 may also import at most one function from M4 with up to 180 dynamic invocations on that function under the executed scenarios (represented by "?F2(180 .. 1) M4"). The matching process will then search the domains of variables to assign functions (as values) to placeholders (as variables) such that the internal constraints (size of the component) and the link constraints (quantity and the call-frequency of the connector links) will be satisfied. The notations "?IF" and "?EF" in the import and export parts denote two unidentified numbers of links between the current component and any other component in the query, such that their interactions have not been constrained by the AQL query. Therefore, "?IR" and "?ER" are not matched by the matching process, however the existing links will be shown in the recovered architecture. In the rest of this section, we present the modeling of the architecture recovery process.

Modeling recovery process as VCSP

Valued Constraint Satisfaction Problem (VCSP) [15] is an extension of the conventional Constraint Satisfaction Problem framework (CSP), that allows us to deal with over-constrained problems. In the VCSP framework, a *valuation* (or cost) is associated with each constraint. The task of assigning a value to a variable in the problem is called an *assignment*. The valuation of an assignment is the aggregation of the valuations of the constraints that are violated by this assignment. The goal in a VCSP model of a problem is to find a complete set of assignments with minimum aggregated valuation. Typically, a search algorithm is used to find an optimal (or sub-optimal) assignment.

Formally, a VCSP framework is defined as a four-tuple P = (V, D, C, f), where V is a set of variables, D is a set of corresponding variable domains, C is a set of constraints between the variables, and f a valuation function (i.e., cost function).

In the adopted VCSP model, the nodes of the source graph, i.e., the functions, are considered as the candidate values to be assigned to variables of the VCSP. The domain of each variable (i.e., a group of eligible functions that can be assigned to a variable) in a component is the same as the domain of the main-seed variable in that component. As we discussed in the static pre-processing the whole search space is divided into domains and the functions in each domain are associated with a distinguished function in that domain called main-seed. During the pattern matching process, in order to recover an abstract component we first choose and assign a main-seed function (as value) to a component's placeholder (as variable). This causes the domain of the main-seed function to become the domain of each placeholder (variable) in that abstract component. This step is performed for each abstract component when that specific component is being recovered. The pattern matching process then searches the domain of the variables to find the best value to variable assignment by minimizing a cost function. A *valuation function* (cost function) consists of three sub-costs: i) *component's internal cost*: determined by the similarities between entities, and static and dynamic function invocations within the component; and ii) *component's interaction cost*: determined by the number of the static function calls and their call frequencies. These two costs will be discussed in the next sub-section.

6. Approximate pattern matching process

After modeling the architectural pattern as an AQL query, a A^* search engine will search for a solution (concrete components and connectors) for the pattern query in an iterative pattern matching process. In this approach, we use a sub-optimal A^* search algorithm [13] to obtain a set of value to variable assignments with minimum cost. The recovery process can be either incremental (i.e., recovery of components one at a time) or automatic (i.e., recovery of

all components in one run). In the rest of this section, we define two types of *similarity constraints* that are used by the A^* search algorithm to evaluate the merit of a function to placeholder assignment in the current component.

Internal similarity constraint

Internal similarity constraint is defined between two functions (values) that are assigned to a pair of placeholders (variables) within a component. This similarity is defined based on "static" *maximal association* between two functions, and the "dynamic" call frequency between two functions in either direction. Maximal association is the property of a group of functions that all share a maximal group of attributes (i.e., called functions, used datatype, and used variables). Maximal association can be obtained by applying data mining operation *association rules mining* on the source graph of the software system [12]. The whole group of functions and their shared attributes is called an *associated group* g_x. We define a similarity metric between each pair of functions in an *associated group* g_x based on the number of shared attributes and sharing-functions.

Formally, static similarity between two functions f_i and f_j, denoted as $sim_{st}(f_i, f_j)$, is defined as the *maximum association degree* between f_i and f_j, considering that f_i and f_j may belong to more than one associated group g_x with a different association degree in each g_x:

$$sim_{st}(f_i, f_j) = max|_{g_x}\ assoc(f_i, f_j, g_x)$$

where:

$$assoc(f_i, f_j, g_x) = |sharedAttributes(g_x)| + \frac{|sharingFunctions(g_x)|}{2}$$

More details about the steps and rationale for the similarity metric $sim_{st}(f_i, f_j)$ can be found in [12]. We intentionally assign a very high internal similarity value constraint between the values (f_i and f_j) of two variables of a component, to be satisfied; therefore, we force that almost all such constraints to be violated. This causes a cost function $cost_{in_{avg}}$ (defined below) to aggregate the average of static and dynamic costs (i.e., $cost_{in}$) of matching between the candidate function f_i (that is being assigned to the current variable) and the functions that have already been assigned to the variables inside that component (i.e., other f_j's). The value of $cost_{in_{avg}}$ is used as a measure of ranking the partially-matched component by the search engine.

$$cost_{in}(f_i, f_j) = 1 - (1 - \frac{1}{4 + freq(f_i, f_j)}) \times sim_{st}(f_i, f_j)$$

$$cost_{in_{avg}}(f_i, \Sigma f_j) = \frac{1}{n} \times \sum_{f_j} cost_{in}(f_i, f_j)$$

where, $freq(f_i, f_j)$ denotes the number of dynamic function-calls between f_i and f_j in either direction, and n is the number of currently assigned functions inside the component. The rationale for defining this metric is as follows. If there is no dynamic interaction between two functions (i.e., $freq(f_i, f_j) = 0$) then $cost_{in}(f_i, f_j)$ is determined by the 75% of the static similarity between f_i and f_j, however if the interaction is very high then $cost_{in}(f_i, f_j)$ is determined by the 100% of the static similarity. Moreover, for low interaction (i.e., $freq(f_i, f_j) = 1$ or so) $cost_{in}(f_i, f_j)$ would decrease with a high rate, but the speed of decreasing would slow down for higher interactions (i.e., $freq(f_i, f_j) >> 1$) providing a reasonable change in the cost function. This behavior is desirable since it reflects the existence of interaction, however high interactions would not predominate the whole distance measure.

External link constraint

An external link constraint is defined between two components C_a and C_b within the AQL query. This link determines the maximum number of static function calls and the maximum number of total call frequencies on these links[1]. This link constraint is considered as a *hard constraint*, i.e., it can not be violated by the pattern matching engine and the violation of such constraint will cause the deletion of the candidate function from the search domain of that component. In other words, the violation of external link constraint will prune the search tree and in the worse case all functions in a variable's domain may be deleted and consequently the pattern matching process fails, or back-tracks to the previous component recovery phase and reassigns the values to the previous component. Therefore, the external link constraint is used to control both the number of static interconnections and the dynamic function invocation traffic between every two components in the architectural pattern defined in the AQL query. This control is easily done by changing the values of parameters for "?Fu(x .. y)" in the IMPORTS or EXPORTS part of the AQL query. Where, "x" is the maximum of total call frequencies among two components, and "y" is the maximum number of the static function call links. Therefore, the collection of "x" and "y" control the overall traffic between two components.

In order to determine the values of "x" and "y" the user first recovers the components defined in the AQL query without considering any link constraints among them. Alborz toolkit will generate a detailed output information about each individual imported/exported function between the components. After investigating the existing interaction

[1]Each function f_i in component C_a may be imported by many functions f_j's in component C_b, each with a different call frequency. In this case, we only consider one import link for f_j and assign the maximum of those call frequencies for that single import link.

traffic between the components, the user will restrict the interaction among the components and run the recovery process once more, where the static and dynamic interactions among the components will be according to the link constraints defined by the query.

Figure 2 illustrates a complicated situation where the assignment of a function to the current variable (placeholder) in component C4, has generated different kinds of connector links. However, in our approach multiple importing (or exporting) of the same function f_j between two components generates only one static link between them. Formally, external link cost $cost_{link}(f_i, C_a, C_b)$ caused by assigning function f_i to a placeholder in either component C_a or C_b is defined as:

$$cost_{link}(f_i, C_a, C_b) = 1 - (\frac{remainAfter}{remainBefore}) \times (\frac{1}{1+callFreqTotal})$$

where *remainBefore* (*remainAfter*) is the difference between the *maximum link quantity* and the number of connector links before (after) matching; and *callFreqTotal* is the sum of the call frequencies of the generated links. Note that we consider the maximum call frequency for a function during the multiple import or multiple export of the same function between two components. This is illustrated in Figure 2 where the currently assigned function to placeholder is exported to component C3 by two connector links with call frequencies 0 and 97, where the call frequency 97 is considered. The rationale for $cost_{link}$ is as follows. After the current function to placeholder assignment, if no link is generated the cost is zero. The cost will increase when more links are generated and the cost is maximum when the maximum number of links are reached by the current assignment (i.e., *remainAfter* = 0). Also, $cost_{link}$ increases when the number of links before assignment is close to its maximum number. However, with the same number of generated connector links, higher *callFreqTotal* will increase $cost_{link}$ more.

Finally, the total cost of assignment is defined as:

$$cost_{total}(f_i, C_a, C_b) = cost_{in_{avg}}(f_i, \Sigma f_j) \; + \; w \times cost_{link}(f_i, C_a, C_b)$$

where function f_j is within component C_b that is linked to the current component C_a. A high value for the weight w generates cohesive components with high dynamic links and low static links, and vice versa.

During the pattern matching process, whenever the search engine assigns a candidate function f_i to a placeholder (variable) in the current component, a cost function $cost_{total}(f_i, C_a, C_b)$ will determine the average similarity between f_i within the component C_a as well as the overall

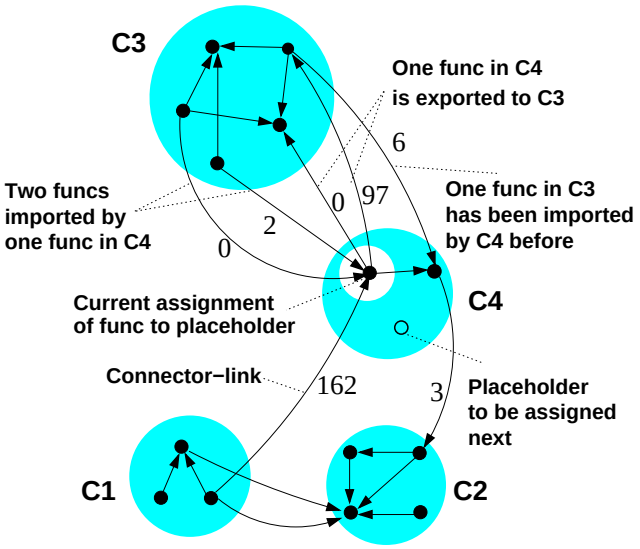

Figure 2. Import/export links and their call frequencies caused by assigning a function to a placeholder during the recovery of component C4.

traffic caused by this assignment in terms of the number of established static and dynamic links that will occur between the current component C_a and a link component C_b.
With the above valuation strategy, the steps for recovery of the components according to the component size and static/dynamic links defined in the pattern query AQL, are described as five steps below.

- **Step 1:** the next variable (placeholder) is selected from the current component C_a to be instantiated.
- **Step 2:** from the domain of this variable the next value (candidate function f_i) is selected to be assigned to this variable.
- **Step 3:** all "internal similarity constraints" and "external link constraints" between the assigned values between two components C_a and C_b are evaluated and checked for satisfaction or violation.
- **Step 4:** the overall cost of the assignment $cost_{total}(f_i, C_a, C_b)$ is calculated. If the cost is higher than the maximum cost of the assignments then the candidate value is discarded, else, the evaluated cost is used as the ranking criterion for the current component and the function is put in the proper place of the list of the assigned functions.
- **Step 5:** the best set of function assignments, i.e., least overall cost of the matching while not violating the link

constraints, for all the variables in the current component is the solution for the valued constraint satisfaction problem.

The result of the recovery process is represented as a concrete component C_a that has import/export relations with the previously recovered components, $C_b, ...$, where the number of the functions in C_a is less or equal to the number of variables that were defined for it in the pattern query. The case of "less" happens when there is no solution (i.e., constraints are not met) with the originally specified number of variables. In the next iteration of the recovery process, the user can define another query to recover a new component of the system, or we can revise the AQL query for the same component and run the pattern matching step again.

7. Case study

In this section, we present the results of applying the proposed approach on Xfig drawing tool [1]. The amalgamation of dynamic information into static information incorporates semantics in the recovery process and hence the whole practice becomes more sensible. In the following we discuss three steps of the proposed architectural recovery framework, including: static pre-processing, dynamic pre-processing, and approximate pattern matching on the Xfig case study.

Static pre-processing
We use Refine C parser [7] to parse the source code of Xfig and generate the source graph using a domain model that restricts the types of the entities in the source graph to functions, data types and global variables. In a further step, we apply the Apriori data mining algorithm [4] on the source graph to produce the associated groups of functions that lead us to generate a similarity matrix consisting of similarity values between every pair of system functions. As mentioned earlier, for each entity in the system a search domain must be generated based on its corresponding similarity values with other entities. To do so, for each entity we group all the entities that have a similarity greater than zero with that entity in its domain. To reduce the time complexity of the search process, very large domains are truncated to a manageable size. This would not affect the recovery process since all functions with high similarity value with the main seed of the domain are kept in a sorted array and only functions with very low similarity values to the main seed are deleted.

Dynamic pre-processing
For the dynamic analysis step, we use GNU gprof profiler [2] to instrument the Xfig source code and capture the execution profiles. Figure 3 presents a set of seven scenarios each with a different combination of several features of the Xfig tool. In order to eliminate the possible effects caused by the execution order of these features, we adopt a set of scenarios that contain different permutations of these features. After executing the scenarios on Xfig system we extract the frequencies for function-calls from the generated execution profiles. The frequency that is assigned to each function-call edge in the source graph is the highest frequency of that edge within the seven generated profiles.

#	Scenario
1	*"Draw, move, rotate, flip, update, edit, scale."*
2	*"Draw, move, rotate, flip, copy, scale."*
3	*"Draw, move, flip, rotate, edit, add text, move point, cut point, change grid, add image, delete."*
4	*"Draw, scale, copy, update, add image, flip, move point, edit, delete."*
5	*"Draw, rotate, flip, update, scale, move point, copy, add point."*
6	*"Draw, rotate, cut point, copy, flip, update, scale, delete."*
7	*"Draw, cut point, rotate, add text, update, edit, scale, add image, change grid."*

Figure 3. Generated scenarios for a particular set of features of Xfig.

Approximate pattern matching
At each iteration of the pattern matching stage, the Alborz tool provides a list of main-seed suggestions. These main-seeds are functions that possess high average similarity values with functions in their domains. The main-seed suggestion algorithm is an approximation of the main A^* search algorithm that computes the average similarity values of the group of highly cohesive functions around the domain's main-seed. The domains are already sorted according to the highest similarity of the functions to the domain's main-seed. In selecting the main-seed for the next domain the overlap of the core parts of the domains are also considered so that the resulting components become highly cohesive and less overlapped. In the static analysis these criteria will produce components that are disjoint, however it is hard to justify the usefulness of these completely disjoint components without any logical relation to each other. The proposed dynamic / static analysis is empowered by feature driven objectives that guide the recovery of logical modules.

Figure 4 illustrates the result of module reconstruction process consisting of four modules M1 to M4 and for two cases *No Link Constraints* and *Link Constraints* that are discussed below. The corresponding AQL query fragment for module M1 has been discussed in Section 5. The number of functions that have been assigned by the search engine to modules M1 to M4 are: 30, 14, 15, and 10, respectively. For each generated module in Figure 4, from top to bottom the

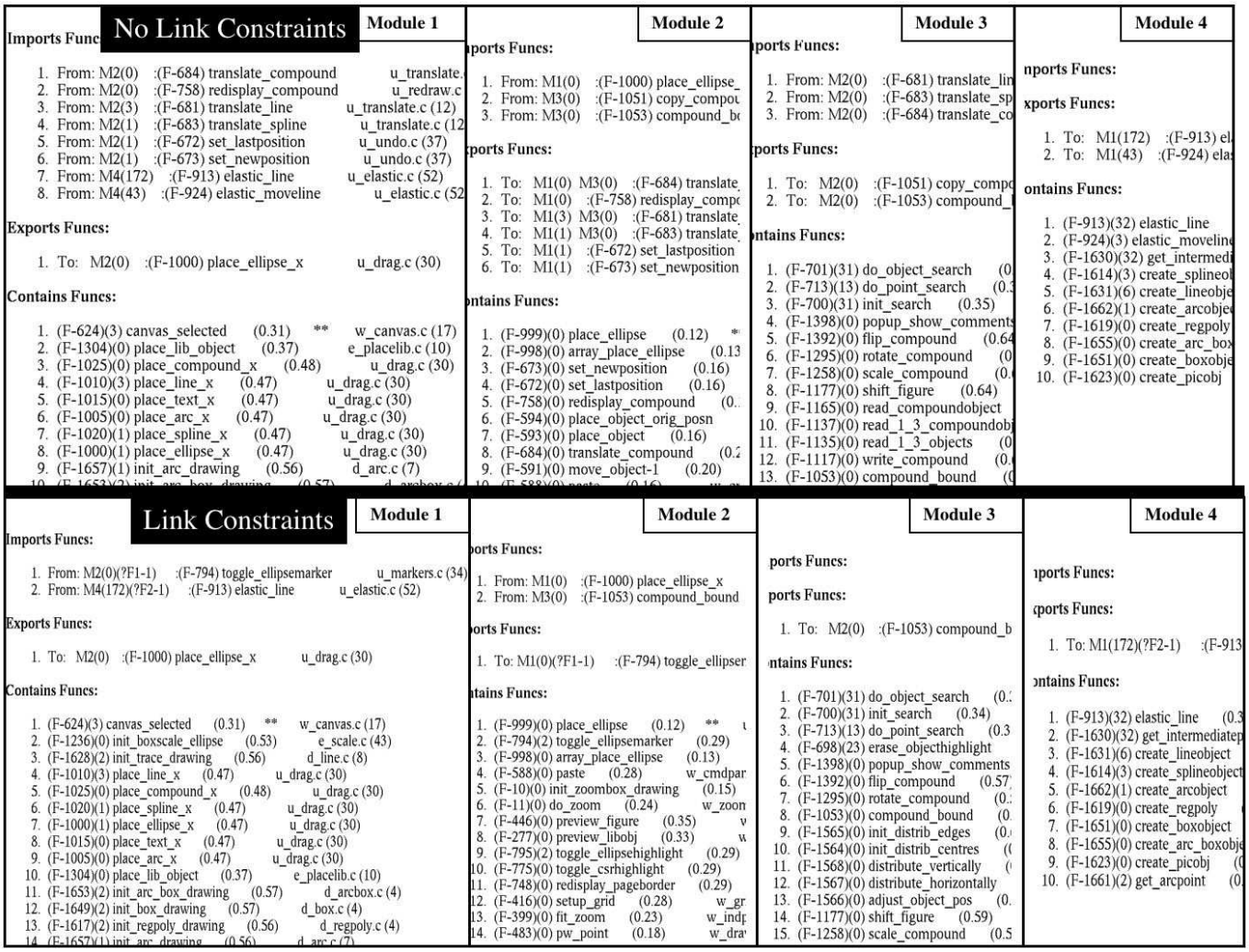

Figure 4. The result of architecture reconstruction process for two cases; i) *NoLink Constraints*, where several dynamic links (import/export functions) exist between Module 1 and Modules 3 and 4; and ii) *Link Constraints*, where both static and dynamic constraints have been applied on the links between the corresponding modules to control their static and dynamic interactions.

following information are presented: imported functions; exported functions, and contained functions. For each part, one function occupies one full line with corresponding information. For example, line 8 in the part "Imports Funcs" of module M1 is repeated below:

8. From M4(43) (F-924) elastic_moveline u_elastic.c (52)

This line indicates that M1 imports function "elastic_moveline" (with code F-924) from module M4 and this function physically located in file "u_elastic.c" with total number of functions 52. Moreover, execution of Xfig scenarios (shown in Figure 3) cause that function "elastic_moveline" to be invoked maximum 43 times by different functions in M1. This is in accordance with the discussion of the example in Figure 2. The interpretation of the lines in the "Contains Funcs" part of a module is presented by using an example in module M1, as follows:

1. (F-624)(3) canvas_selected (0.31) ** w_canvas.c(17)

Function "canvas_selected" is a main-seed (because of sign "**") and has an average similarity value 0.31 to the other functions in M1, and has been invoked by a maximum frequency 3 by the other functions in M1.

The generated modules have sufficient information to provide the user with a deep insight into the software system and the quality of the recovered architecture. As discussed earlier in the proposed environment in Section 3 the user first runs the module reconstruction process with no link constraints in order to generate high cohesive and high internal dynamic interaction components. This allows the user to investigate the static and dynamic interactions among the generated components and to improve the quality of the obtained components to be less dynamically or statically inter-dependent. In the next step, the user employs

the link constraints on the import / export parts of the AQL query to limit the static and dynamic interactions among the components. The VCSP search engine then identifies a new module configuration that satisfies the enforced constraints (if possible). The VCSP search operation will backtrack to the previously recovered modules to revise the solutions of the previously found modules as an attempt to generate a solution for the current module in tightly constrained situations. The "Link Constraint" part of Figure 4 demonstrates the result of module reconstruction after constraining the links between modules M1 with modules M3 and M4 as discussed in AQL query earlier. A comparison between the recovered modules in two cases of Figure 4 indicates how VCSP search has met the AQL constraints. The link constraint between M1 and M2 (i.e., "?F1(0 .. 4) M2") has caused that the four imported functions with dynamic interactions (in No Link Constraints results) to disappear in the recovery with Link Constraints situation. Also the link constraints between M1 and M4 (i.e., "?F2(180 .. 1) M4") has caused one of two imported functions to be rejected after applying the constraints, however, in this case the dynamic interaction (i.e., 180) was not a restricting factor.

Therefore, both static and dynamic *Link Constraints* have been applied on the links between the corresponding modules to control their static and dynamic interactions, hence producing modules that are best suitable for deploying over the internet for distributed computing applications.

8. Conclusion

In this paper, we introduced a novel method of amalgamating static information with run-time dynamic information in a pattern-based architectural recovery technique. The static information is the foundation of the approach, where the dynamic information adds semantics and focused information to the whole practice through embedding profiling of frequently used features. A seamless integration of these two rather orthogonal types of information while challenging is very fruitful reverse engineering activity. In this respect, the proposed approach attempted to contribute in different ways, including: reducing the inherent complexity of the pattern matching search by reducing the search domain size; focusing on the essential parts of a large software systems; controlling the dynamic interactions among components as a means to leveraging distributed computing properties of the software. In order to inject dynamic information into our pattern matching engine, we run a number of frequently used task scenarios on the software system and recorded the dynamic function invocation frequencies for different system functions. These dynamic information were further embedded into the source graph edges and consequently were incorporated into measuring the overall cost function of the pattern matching process. Finally, the whole recovery process has been modeled as a Valued Constraint Satisfaction Problem (VCSP), where functions represent values and component placeholders represent variable. Through a case study, we presented how controlling the dynamic interaction among recovered components is sensible in restructuring a software system.

References

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

[2] Gnu gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

[3] Odyssey project. http://reuse.cos.ufrj.br/site/.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 487–499, 1994.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[6] M. A. Eshera and K. S. Fu. A similarity measure between attributed relational graphs for image analysis. In *Seventh International Conference on Pattern Recognition*, pages 75–77, 1984.

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

[8] T. Richner and stephane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of IEEE ICSM'99*, page 13, 1999.

[9] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the IEEE CSMR*, pages 47–55, 2002.

[10] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *In Proceedings of the IEEE IWPC'01*, pages 115–116, Toronto, Canada, May 2001.

[11] K. Sartipi, N. Dezhkam, and H. Safyallah. An orchestrated multi-view software architecture reconstruction environment. In *Proceedings of IEEE WCRE'06*, pages 61–70, October 2006.

[12] K. Sartipi and K. Kontogiannis. Component clustering based on maximal association. In *Proceedings of WCRE'01*, pages 103–114, Stuttgart, Germany, October 2001.

[13] K. Sartipi and K. Kontogiannis. On modeling software architecture recovery as graph matching. In *Proceedings of ICSM'03*, pages 224–234, 2003.

[14] K. Sartipi, L. Ye, and H. Safyallah. Alborz: An interactive toolkit to extract static and dynamic views of a software system. In *Proceedings of the ICPC'06*, page to appear, June 2006.

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

[16] A. van Deursen, et al. Symphony: View-driven software architecture reconstruction. *WICSA'04*, pages 122-132, 2004.

[17] A. Vasconcelos, et al. An approach to program comprehension through reverse engineering of complementary software views. In *PCODA'05*, pages 58–62, USA, 2005.