Multi-view Software Architecture Reconstruction

Multi-view Software Architecture Reconstruction using Design, Dynamic, and Static Analyses

By
Nima Dezhkam

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Masters of Applied Science

McMaster University

DEGREE:	MASTERS OF APPLIED SCIENCE (2006)
DEPARTMENT:	Computing and Software
University:	McMaster University, Hamilton, Ontario
TITLE:	Multi-view Software Architecture Reconstruction
AUTHOR:	Nima Dezhkam, B.Sc.
SUPERVISOR:	Dr. Kamran Sartipi

NUMBER OF PAGES: x, 87

Abstract

Most approaches in the reverse engineering literature generate a single view of a software system. However, a single view recovery restricts the scope of the reconstruction process to limited types of information. In this thesis, we propose a multi-view approach that recovers three views of software systems: design, behavior, and structure. The design view is reconstructed through transforming a number of task scenarios into design diagrams (class diagrams, ER diagrams, and activity diagrams) using a novel scenario domain model that allows us to parse the task scenarios and populate an objectbase of actors and actions. The behavior view is represented through a set of profiles that contain the dynamic information extracted from executing a set of relevant task scenarios on the software system. This set of task scenarios covers frequently used software features. The obtained profiling information serves as the dynamic characteristics of the software system that would be embedded into the structure view recovery. Finally, we propose a pattern based structure view recovery that defines the high-level architecture of the software system using abstract components and interconnections. In this context, both static and dynamic aspects of the software system are used to collect software entities into cohesive components with reduced dynamic interactions. The whole process is modelled as a Valued Constraint Satisfaction Problem (VCSP). As a case study we applied the proposed approach on the Xfig drawing tool with promising results.

Acknowledgements

I would like to express my sincere and gratitude to all those who gave me the possibility to complete this thesis. I am deeply grateful to my supervisor Prof. Kamran Sartipi whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing of this thesis. His overly enthusiasm and integral view on research and his mission for providing high-quality work has made a deep impression on me. Besides of being an excellent supervisor, he was as close as a relative and a good friend to me.

I would like to thank my co-supervisor Prof. Mark Lawford who kept an eye on the progress of my work and always was available when I needed his advises. I appreciate my official M.A.Sc. defense committee Prof. Rida Khedri and Prof. Sanzheng Qiao for their constructive criticism and excellent advice.

I wish to thank all my colleagues, both past and present in the Department of Computing and Software in McMaster University for their assistance during the progression of this research. I also would like to thank my kind friends who supported me throughout my studies and research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems and their platforms become costly to maintain after being used for 10-15 years. There exist several reasons for this incident, such as: lack of updated documentation; error-prone operation caused by patches and feature improvements; cease of platform support from the provider; and adopting new technologies (plug-ins or inter-operability techniques). In this context, the target system would turn into a legacy system where in most cases the organizations are forced to perform a maintenance (reverse / re-engineering) operation to keep their software operational rather than replacing the system with a new one. This is mainly because the replacement of the legacy systems is very expensive in terms of required budget and time. Having a large number of users that have operated the company's business for several years with a single system, makes the replacement of the system infeasible in some cases. In this context, software maintenance should be performed to make the legacy system meet their changing business requirements. In this regard, having a good understanding of the system is crucial in maintaining the software assets.

1.1 Software architecture recovery

Software architecture is a valuable type of information about the system that increases its understandability. There is no single definition for software architecture. One of the major definitions which is proposed by Software Engineering Institute (SEI) is as follows:

"Software architecture describes the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."[4]

Consequently, software architecture recovery is considered to be a major activity in software reverse engineering for the sake of software maintenance.

Software architecture recovery refers to extracting the description of system components and their relationships from a low-level software representation such as source code that can be used for a software maintenance activity [10].

In this regard, different architecture recovery techniques have been proposed that can be categorized as follows:

- *Clustering* techniques that try to group together system entities into clusters based on their similarities.

- *Concept lattice analysis* techniques that consider a relation between the objects in the system and their attribute values and visualize the structure of their relations in a lattice.

- *Pattern based* techniques that define a high-level architecture (pattern) for the system and try to find the defined pattern in the source code.

- *System visualization and analysis* techniques that allow one to define and visualize the hierarchy of the subsystems in a software system based on the dependencies between system entities.

1.2 Problem statement

There are a large number of approaches to software architecture recovery that focus on a single view of the system. However, a single view approach is restricted to the limitations of that view. For example, a static analysis approach lacks the run-time information about the system. Static analysis approaches consider only the "existence" of data/controls dependencies among software components and stay silent about dynamic information such as frequency of such dependencies being used in the runtime execution of the system. The interactive and dynamic nature of component-based applications, and the importance of minimizing inter-component traffic necessitate the dynamic analysis of dependencies between software entities and leveraging the result of this analysis in combination with static analysis information to enhance the component clustering practice. In general, different views of a software system are the result of applying separation of concerns on a software engineering activity such as software development or reverse engineering. There are a number of techniques proposed in the area of forward engineering that approach software development from different software views. Zachman's framework [57] and Krutchen's 4+1 views [27] are two examples of such approaches. However, in the reverse engineering context, recovering the software views, such as design, behavior, and structure views, of a legacy system is much more challenging than forward engineering because of the poor documentation and un-tracked changes made to the system. The structure view, which is recovered based on static analysis techniques, is the view most commonly recovered in the literature and large number of techniques and tools have been proposed to address this issue [26, 29, 20, 49, 45, 55]. Dynamic analysis techniques, as another category of techniques, have been widely used to recover the behavior view of the system which is based on the run-time properties of a software system. A variety of techniques and tools are also proposed for dynamic analysis approaches [14, 15, 8, 32, 53, 17]. Recently, the reverse engineering research community has paid more attention to amalgamation of static and dynamic aspects of software systems with the goal of taking advantage of dynamic analysis (e.g., profiling, or dynamic pattern recovery) to enhance the results of the static analysis [51, 6, 36, 35]. Nevertheless, static analysis at low or high levels of abstraction (e.g., source code analysis, or architectural recovery) is still considered as the main focus of maintenance activities. One of the main reasons for this is the completeness of the static information about the system with regard to any objective for analysis, as opposed to that of dynamic analysis which is based on execution of a limited number of task scenarios. On the other hand, dynamic analysis provides a link between the software functionalities (i.e., software features) that are represented by task scenarios and the source code elements that implement those functionalities. This valuable information usually can not be obtained easily in a static analysis task.

In addition to the aforementioned two views, the design view, as the third view, provides a high-level representation of software artifacts and their dependencies which allows for conceptual understanding of the software system. The design view, when integrated with structure and behavior views, provides a link between static and dynamic analyses and the high-level design representations of the system which are easily understandable by humans.

Based on the above discussion, this thesis defines the multi-view software architecture recovery problem as:

devising the process, required techniques, and supporting tool to tackle the limitation of the information extracted from single views of a software system and relating *the views through user interaction with the system.*

1.3 Proposed approach

In this thesis, we propose a multi-view software architecture recovery approach that generates three views of a system: design, behavior, and structure. In this approach, a systematic generation of the task scenarios derived from the existing evidences followed by a schema-based scenario-to-design transformation process generates the design view of the software system. Furthermore, the analysis of the run-time execution profiles that are the result of executing a set of specific scenarios on the system lead to behavior view recovery. Finally, the analysis of the data/control dependencies can yield the structure view of the software system. This multiple view analysis provides deep insight into the design properties of the implemented software system, and serves as a means to enhance the system's architectural design.

In this work, the multi-view framework is supported by the *Alborz* toolkit [38] built within the Eclipse plug-in environment.

1.4 Contributions of this research

The major contributions of this research are: i) proposing a multi-view framework that recovers three views of software system; deign, behavior, and structure, and combines these views with guidance of task scenarios; ii) proposing a novel scenario domain model to parse the text of scenarios and decompose them into design diagram ingredients; iii) enhancing the Alborz [37] static architecture recovery tool by combining static and dynamic information of the system in order to recover less dynamically interactive components; and iv) providing a case study that demonstrates the results of applying the proposed multi-view technique on a medium-sized software system, the Xfig drawing tool [3].

1.5 Organization of the thesis

The rest of this thesis is organized as follows:

Chapter 2 provides a review on the related work for our approaches for design, behavior, and structure recovery along with related multi-view reverse engineering approaches.

Chapter 3 presents formal representations of our multi-view technique.

In Chapter 4 the proposed multi-view frame work is introduced.

Chapter 5 presents our scenario-based design recovery approach.

Our combined static and dynamic architecture recovery technique is presented in Chapter 6.

Chapter 7 provides a case study applying our multi-view technique to the Xfig drawing tool.

Finally, Chapter 8 presents some concluding remarks.

Chapter 2

Related work

The proposed research in this thesis is related to the approaches in three views of the software architecture that are discussed separately in the remainder of this chapter.

2.1 Structure view recovery

The literature for structure recovery in this thesis refers to approaches for pattern-based software architecture recovery that try to recover architectural patterns in the software system and use an instance of the Constraint Satisfaction Problem (CSP) to evaluate the recovered solutions. A solution to the CSP is an assignment of values to varibales of the problem such that constraints are satisfied. One of such approaches is proposed by Woods [54], where he uses a search algorithm to find the solution. Similarly, in our approach we define architectural patterns to be recovered. However, we model the recovery process as an instance of the Valued Constraint Satisfaction Problem (VCSP), which is an extension to the CSP that allows for violation of constraints with some cost. Similarly, in our approach we use a search algorithm to find a solution to the problem, which is an assignment with minimum cost.

In [43] an approach to software architecture recovery is proposed that models the recovery as an instance of VCSP where again each constraint violation has a cost. Similar to our approach in this thesis, the cost of a value assignment is calculated based on cost of edges between the entities of the source code which has an inverse relation with the similarity of the entities. However, in contrast to the approach in [43], we define architectural patterns that guide the structure recovery.

Kazman and Bruth [25] propose an interactive architecture pattern recognition technique to recover user-defined patterns of system structure. They define the pattern as a graph of system elements and use the CSP to match the defined pattern with the system entities. The patterns in their approach describe the interaction between individual elements of the system, as opposed to our approach that the pattern defines an overall set of constraint for the modelled component.

2.2 Scenario-based design view recovery

The proposed scenario-based design generation approach relates to the literature for capturing and using task scenarios for design related activities. Hufnagel et al. [52] present a scenario-driven object oriented requirements analysis for documenting the design of a system. However, in contrast to our approach they do not define a scenario schema and their approach is methodology dependent. In [33] a method for modular representation of the scenarios is proposed that supports the reusability of the scenarios in different design contexts. This approach is similar to ours in the sense that it attempts to define a structure for the scenarios. In [23, 24] Kazman et al. define a software architecture analysis method (SAAM) where the important task scenarios in an application domain are mapped onto the architectural representation of competitive software systems in that domain to evaluate and compare their architectural design qualifications. However, in SAAM the scenario to architecture mapping is subjective and the scenarios are not structured, whereas we define structured scenarios that assist more accurate mapping of scenarios to design information. In [12] Potts defines a schema for semantic models of scenarios to help with requirements refinement of a software system. In contrast, the scope of scenario schema in our approach is extended to cover the design aspects of a system.

2.3 Behavior view recovery

The proposed approach for behavior recovery in this thesis relates to the approaches that use dynamic analysis to capture and analyze execution traces of a software system and extract some behavioral properties of the system. Eisenbarth et al. [14, 15] use concept lattice analysis on the execution traces of a system resulting from a set of scenarios, to locate computational units that implement certain features of the software system. Similarly, in our approach we use the relation between scenarios and computational units (i.e., functions) that are invoked during the scenario execution.

Wilde et al. [53] propose a set difference approach to execution traces for locating software features where the set of functions in the related scenario executions are differentiated in order to localize the implementation of a specific feature. In our approach, we also use the notion of feature-specific scenarios, however, we extract frequencies of function-call execution as evidences of the feature implementation and structural modularity evaluation.

In a different context, El-Ramly et al. [17] applied a sequential pattern mining technique to find interaction patterns between graphical user interface components. Also, in the work of Zaidman [58] a web-mining technique is applied on program dynamic call graphs that supports the program comprehension. Similar to our approach, the above approaches analyze the execution traces to extract a summarized behavior information; however, in our approach we aim for extracting frequencies of calls rather than mining patterns of execution.

2.4 Multi-view recovery

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

Vasconcelos et al. [51] present a dynamic analysis-based 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. The extracted views in their approach complement the static view through integration with a tool set already integrated into a reuse based software development environment, named Odyssey [6]. Similar to our approach, they use dynamic analysis to recover the behavior view of the system along with complementary views.

Riva et al. [36] propose a technique for architecture recovery using combined static and dynamic information. Their technique is based on choosing architectural concepts and applying abstraction techniques on source code to manipulate the concepts at the architectural level. 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. [50] propose a view-based software reconstruction framework that provides a common framework for reporting reconstruction experiences and comparing reconstruction approaches.

Richner et al. [35] propose an approach to extract static and dynamic views from Java programs. The static view is generated from class files and visualized using the

Rigi reverse engineering environment [2]. The dynamic view which is represented as scenario diagrams, are attached to the static Rigi graph. The overlapping information between two views forms a connection for information exchange between the views. Similarly, in our approach we use common information, such as scenarios, to relate the recovery of different views.

Chapter 3

Formal representations of different views

The proposed multi-view framework in this research extracts three views, namely design, behavior, and structure, of a software system by applying three different functions to the system. Other than extracting three views in our framework, these views are integrated through common information. Scenarios are the core means for generating the design and behavior views and integrating these two views. On the other hand, the frequencies that are found for source code function-calls in the behavior view recovery will be embedded in the source graph of the structure view and used in the pattern matching process. In this chapter, we define the overall process of the proposed multi-view framework by formal notations.

If we name the group of system and its available documents , such as: executable code, source code, and requirements, as Sys then we can define three functions to recover the three views as follows:

Design view = $\pi_{design} Sys$,

Behavior view = $\pi_{behavior} Sys$, and

Structure view $= \pi_{structure} Sys$

where π_x denotes a specific function for view x.

In the rest of this chapter, we explain each of the functions mentioned above.

3.1 Design view

Scenarios are the major building blocks of the design view. In this research we use English text to present scenarios. In Chapter 5 we define a syntax for the structure of scenarios to control the structure of the English text. We say that each scenario that conforms with this structure has type $Scenario$.

In this research we generate three types of design diagrams that can be categorized into *data*, *function*, and *network* diagrams. A set of scenarios that are defined for the system determine the ingredients of the design diagrams. Data diagrams focus on the data items that are manipulated in fulfilling the set of scenarios along with the relationships between data items [1]. Function diagrams represent the sequence of actions that should be taken to perform the set of scenarios [2]. And finally, network diagrams depict the physical locations of the system that the scenarios are performed in [3].

Regarding the above discussion, the π_{design} function that recovers the design view is defined below. Here we let $\mathbb{P}$ denote the power set operator.

$\pi_{design} Sys : [\mathbb{P}(Scenario) \rightarrow Data\text{-}diagram \times Function\text{-}diagram \times Network\text{-}diagram]$

$Data\text{-}diagram : [\mathbb{P} Data] \times [\mathbb{P} Data\text{-}dependency] \times [\mathbb{P} Constraint]$

$Function\text{-}diagram : [\mathbb{P} Action] \times [\mathbb{P} Action\text{-}dependency] \times [\mathbb{P} Constraint]$

[1]We use conventional diagrams, such as Entity-Relationship and class diagrams, as our data diagrams.

[2]We use diagrams such as function diagram and activity diagram for this purpose.

[3]We use a node and interconnection representation to illustrate network diagrams.

$Network\text{-}diagram : [\mathbb{P}Action] \times [\mathbb{P}Action\text{-}dependency] \times [\mathbb{P}Constraint]$

where *Data*, *Action*, *Data-dependency*, *Action-dependency*, and *Constraint* types are classes of the scenario domain model that is discussed in detail in Chapter 5. $\mathbb{P}Scenario$ is the type of a set of structured scenarios A that can be generated for a system S of type S_type.

3.2 Behavior view

Behavior view is the result of projecting out a number of runtime attributes of a software system. To run and study a system in a reasonable manner, a set of scenarios should be executed on the system. Since in this research we are interested in exploring a system (or a part of it) from different views, to unify the scope of our observations in design and behavior views we take the same scenario set A that was used in the design view generation as the reference to generate a new set of scenarios $A\prime$ for behavior view recovery. The scenarios in this new set cover a set of features F, called the set of frequent features. More detailed definitions of features and frequent features are presented in Chapter 6.

$F = \{f_1, .., f_n\}$ $set\ of\ frequent\ features$

$A\prime = \{A\prime_1, .., A\prime_m\}$ $set\ of\ scenarios\ that\ cover\ features\ in\ F$

After executing the scenarios in $A\prime$ on the instrumented (profiling-enabled) system, the generated execution profiles yield all the executed functions and function-calls for each scenario along with the number of their occurrence (*count*). A function-call is defined as an edge from a *caller* function to a *called* function. Therefore we define a set O and a set E for each of the executed scenarios A_x in $A\prime$ as follows:

$O_x = \{(o_1, count_1), .., (o_n, count_n)\}$ set of executed functions along with their number of executions

$E_x = \{(e_1, count_1), .., (e_m, count_m)\}$ set of invoked function-calls along with their number of invocations

After obtaining the O and E sets for all the scenarios in $A\prime$, a frequency $freq$ can be derived for each function o_i and function-call e_j in the system based on the set of executed scenarios $A\prime$ as follows:

$$freq_{o_i} = \sum_{A_x \in A\prime} count_i \ where \ (o_i, count_i) \in O_x$$

$$freq_{e_j} = \sum_{A_x \in A\prime} count_j \ where \ (e_j, count_j) \in E_x$$

Therefore, if we show the frequencies using natural numbers ($\mathbb{N}$), the behavior can be represented as follows:

$\pi_{behavior} Sys : [\mathbb{P}Scenario \rightarrow [Function \rightarrow \mathbb{N}] \times [Function\text{-}call \rightarrow \mathbb{N}]]$

where *Function* and *Function-call* are types of a group of entities and relationships in the source code respectively, which will be defined in Chapter 6. Simply, a *Function-call* can be considered as an edge between two *Functions*. Consequently, we can define two functions $funcFreq$ and $callFreq$ as representatives of behavior view as follows:

$funcFreq : \mathbb{P}Scenario \times Function \rightarrow integer$

$callFreq : \mathbb{P}Scenario \times Function\text{-}call \rightarrow integer$

These functions return the frequencies of invocation of a specific function or function-call, given a set of scenarios as input.

3.3 Structure view

In the structure view, to facilitate further analysis, the source code of the system is transformed into an attributed graph notation G^s, called the *source graph* that is briefly defined as follows:

$G^s = (N^s , R^s)$,

$N^s : \mathbb{P}(Entity)$, and

$R^s : \mathbb{P}(Entity \times Entity)$

where $R^s \subseteq N^s \times N^s$ and $Entity$ can be of one of the types of $Function$, $Variable$, or *Data-type* (that refer to type of functions, variables, and data types in the source code respectively).

As mentioned above, the nodes $n_i \in N^s$ represent files, functions, data types, and variables and the edges $e_j \in R^s$ represent relationships between nodes, such as *function-call*, *variable-use* and so on. These nodes and edges have a number of attributes, such as *name*, *id*, and *frequency.* The list of attribute-value tuples can be accessed as a record using function Ψ.

In the presented approach, a conceptual architecture (architectural pattern) is specified by the user in the form of a query consisting of a set of conceptual components, and a set of corresponding main-seeds and constraints using a standard markup language, such as XML. After the pattern query is provided for the tool, the pattern matching algorithms tries to extract a concrete architecture from the system structure (source graph) that conforms with the given conceptual architecture and corresponding constraints. The recovered concrete architecture, after the pattern matching, is represented by a set of *modules* and *import, export* dependencies.

Every module (component) is obtained from grouping a number of entities that have a high *similarity* with the set of main seeds specified for that module. More formally, we define a module M with a set of main seeds S as:

$S : \mathbb{P}Entity$, and

$$M = \{(N^M , R^M) \mid N^M \subseteq N^s \wedge R^M \subseteq R^s \wedge R^M = R^s|(N^M \times N^M) \wedge \forall n_i \in N^M$$
$$n_i \; has \; a \; high \; similarity \; with \; main \; seed(s) \; S \; of \; M\}$$

where "|" denotes restriction.

Every *import* relation represents a module using an entity that is out of its corresponding entity set (N^M), and every *export* relation represents a module providing an entity of an *importer* module. Hence the type of these two relations are as follows:

$import : import\_type = M \times Entity$, and

$export : export\_type = M \times Entity$

The syntax of XML pattern queries is enforced by an XML Schema Definition (XSD) presented in Chapter 6. We call every pattern query that conforms with this schema of type $Query$.

With above discussion, the function for structure view recovery, $\pi_{structure}$, can be defined as follows:

$\pi_{structure}Sys : [G^s \times \mathbb{P}Query \rightarrow \mathbb{P}M \times \mathbb{P}import\_type \times \mathbb{P}export\_type]$

As it was mentioned earlier, design and behavior views are integrated by sharing a common set of scenarios as their reference. On the other hand, behavior and structure views are connected to each other through the frequency attribute of the functions and function-calls in the source graph. Assuming that the behavior recovery is based on a set of scenarios named $A\prime$, the integration of behavior and structure views can be formulated as follows:

$\forall function : Function \in N^s \wedge \ \forall funcCall : Function\text{-}call \in R^s :$

$\Psi(function).frequency = funcFreq(A\prime, function) \wedge$

$\Psi(funcCall).frequency = callFreq(A\prime, funcCall)$

where $funcFreq$ and $callFreq$ functions were defined in Section 3.2.

Chapter 4

Proposed multi-view framework

The proposed framework for multi-view architecture recovery is illustrated in Figure 4.1. This framework presents the overall mechanism to extract three views of a software system. As shown in this figure, the whole mechanism can be divided into three major processes; design view recovery, behavior view recovery, and structure view recovery. In this chapter each of these processes is introduced and discussed briefly. The detailed discussion of each view is provided in separate chapters.

4.1 Design view recovery

In the design recovery process, a set of task scenarios are generated using the evidences derived by the user's knowledge of the application domain, system-user interaction, available high-level system documents, and user manuals. The scenarios are parsed to generate a design view of the software system that is represented by two type of diagrams, *entity-relationship diagram (E-R)* and *activity diagram* that represent the implemented functionality and the major system data that are manipulated by the activities. Design view recovery is discussed in detail in Chapter 5.

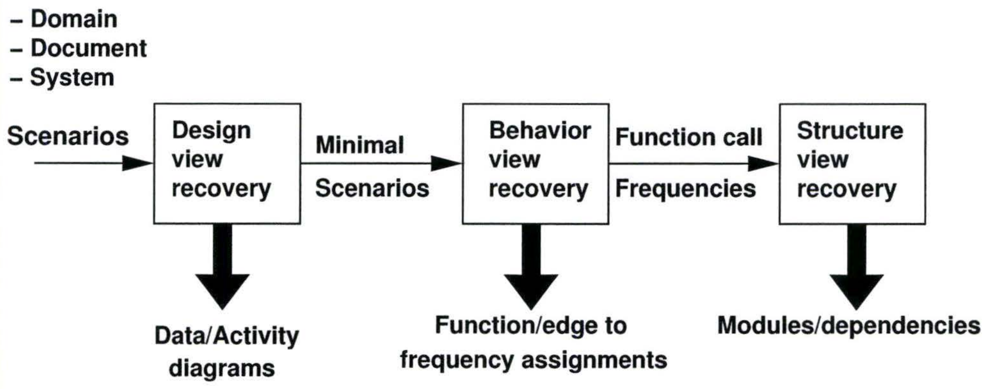


Figure 4.1: Proposed multi-view process to extract three views of a software system.

4.2 Behavior view recovery

In the behavior view recovery process, the user investigates the scenarios in the design view to recognize the features covered in them and then selects a group of frequently used features. This leads to generation of a new set of scenarios that cover the frequent features. The execution of this new set of scenarios on the *instrumented* [1] software system generates execution profiles. The profiles are analyzed and as a result the source code functions and function-calls that were invoked in the execution of the scenarios along with their frequency of call is obtained. These frequencies are then embedded in the attributes of the source code artifacts to be used in the structure recovery stage.

[1]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.

4.3 Structure view recovery

In the structure view recovery, a conceptual architecture is defined as a pattern and then the tool tries to find the specified pattern in the source code using a pattern matching technique. The whole process is modelled as a Valued Constraint Satisfaction Problem (VCSP). The behavior recovery and the structure recovery processes are discussed in detail in Chapter 6.

Chapter 5

Design view recovery

5.1 Introduction

Scenarios-based requirement analysis has attracted significant attention within the requirement engineering field [46]. Scenarios are represented in a variety of formal and informal methods ranging from simple text and graphical media to relational algebra [13]. In this paper, we define a scenario as "a structured narrative text describing a system's requirements in terms of system-environment interactions at business rule level". Scenarios are considered as easy-to-use and effective means in different phases of software engineering process, such as: requirement elicitation and analysis, design representation, code development, testing, and maintenance [31, 22, 30, 47]. Two major issues in scenario-based requirement engineering, *completeness* and *consistency* checking of a set of scenarios, are considered to be challenging tasks. In this context, formalizing the representation of scenarios is so far considered as a solution to this problem [60, 13]. In addition to formal representation of scenarios, there is a wide range of research in requirement engineering domain investigating: the enhancement of scenario generation by using scenario schemas or pre-defined structures [12, 59]; scenario analysis and knowledge extraction [11]; and design-related document generation [48, 52]. However, considering the complexity involved in the aforementioned approaches, it is desirable to devise a technique that allows one to enhance the structure of the text-based scenarios in order to assist design document generation.

In this chapter, we introduce a novel technique to transform the information from scenarios into well-formed design diagrams. In this technique scenarios are generated using domain knowledge and in conformance with a regular expression syntax that imposes a structure to the scenario representation. Further, the generated structured scenarios are parsed using a novel *scenario domain model* to populate an objectbase of design related entities and dependencies. The proposed approach allows one to reuse the domain knowledge and business rules within the scenarios thorough a scenario template knowledge base. The populated objectbase serves both as a data source during the design diagram construction and as a valuable electronic asset of design knowledge to be analyzed, augmented, and used during the maintenance phase of the software system.

At the end of this chapter, as an example application of our design view recovery technique, a case study of a fast-food restaurant system is presented.

5.2 Proposed framework for design view recovery

In this section, we discuss the steps for transformation of the knowledge embodied in the text of scenarios into design related information in three types of design diagrams (data, function, and network) using a framework that is illustrated in Figure 5.1. In a nutshell, the proposed approach generates a set of structured scenarios and uses a domain model to parse these scenarios into ingredients of the view-based design

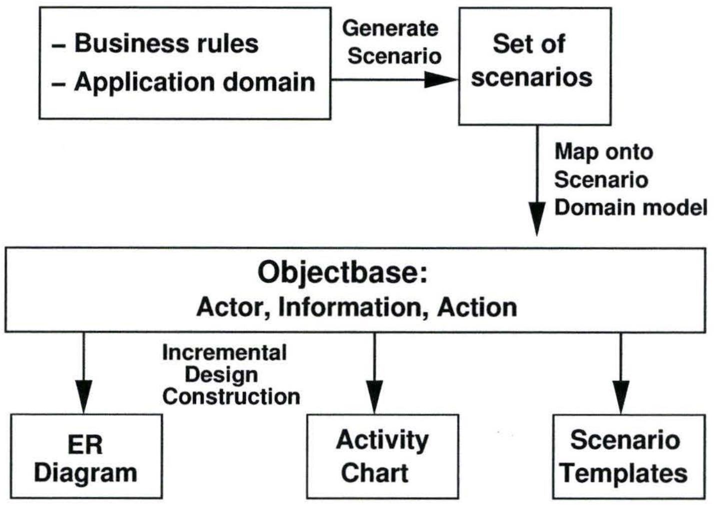


Figure 5.1: The proposed design construction framework from scenarios.

representations. The proposed framework consists of three stages, as follows.

Stage 1 *(scenario generation)*: This stage consists of structured scenario generation and syntax conformance steps. To facilitate scenario generation and control the format and vocabulary of suggested scenarios, a pre-defined set of domain-specific templates can be used. Consequently, at the end of this stage, a set of qualified scenarios that cover a part or the whole of the requirements of the system is achieved. This stage will be discussed in Section 5.3.

Stage 2 *(scenario decomposition)*: In this stage the qualified scenarios are mapped onto the proposed scenario domain model in Figure 5.3. This domain model is a means to parse the structured scenarios and generate instances of classes Goal, Actor, Working information, and Action, and their corresponding dependencies that are defined in the scenario domain model. The generated instances incrementally populate an objectbase of design information that is used to generate design-related diagrammatic representations. This stage is discussed in Section 5.4.

Stage 3 *(design diagram generation)*: This stage deals with generating design diagrams from the information stored in the objectbase resulted from Stage 2. To achieve this, we follow a set of view-specific guidelines to incrementally construct different design diagrams, such as: entity-relationship (ER) and class diagrams for data; activity and function diagrams for function; and node and interconnection diagram for network. The incrementally constructed design diagrams allow for detecting potential inconsistencies between the pieces of design documents which can be tracked back to the requirements and the corresponding scenarios. This stage is discussed in Section 5.5.

5.3 Scenario generation (stage 1)

In this research, we adopt a structured text-based representation for scenarios that conform with a scenario structure syntax. These scenarios will be further used to populate a categorical knowledge-base (representing the knowledge domain) to reuse the captured business rules in a future similar case.

Scenario structure

We define a structure for the generated scenarios that is imposed by a regular expression syntax as follows:

$$Scenario : \{Actor + \{Constraint\}^{0..N}\}^{1..M} + \{Action + \{Constraint\}^{0..N}\}^{1..M} + \{Working\ information + \{Constraint\}^{0..N}\}^{1..M}$$

where "+" and "0..N" represent composition and range, respectively.

The semantics are defined by the application domain's business rules. In this form, each scenario is composed of instances of the classes in a scenario domain model that define the types of entities and relationships in the corresponding application domain.

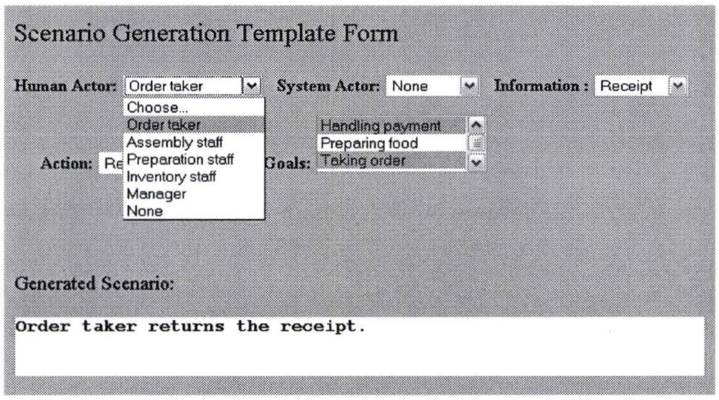
In this scenario syntax the entities *Actor, Action*, and *WorkingInformation* are the entity-types and action-types that will be defined in Section 5.4. Each scenario consists of a sequence of one or more *Actors*, *Actions*, and *Working Information*, each of which can have zero or more *Constraints*. In this form we can generate syntactically correct scenarios which will be further decomposed to populate the objectbase in Section 5.4 and generate design diagrams in Section 5.5.

Task scenario templates

In order to facilitate reuse of the captured domain knowledge and business rules, the proposed framework populates a knowledge-base of scenario templates which are organized to store the structured scenarios for a specific application domain. The idea is to allow a software engineer to assemble task scenarios for the subject software system from relevant application domains. Figure 5.2 illustrates a sample scenario template form for a fast-food restaurant system. This form consists of fields such as: Actor, Information, and Action, where each field possesses a vocabulary of corresponding business terms. The generated scenario at the bottom of the form is a proper composition of the terms selected from these fields.

5.4 Scenario decomposition (stage 2)

The class diagram representation of the proposed scenario domain model is presented in Figure 5.3. This domain model is intended to cover the potential information types (classes) in scenarios from different application areas. For example, we applied this model on three systems, including a software analysis tool "Alborz toolkit" [37], a


Scenario Generation Template Form

Human Actor: Order taker
System Actor: None
Information: Receipt

Choose...
Order taker
Assembly staff
Preparation staff
Inventory staff
Manager
None

Action: Re

Goals: Handling payment
Preparing food
Taking order

Generated Scenario:

```
Order taker returns the receipt.
```

Figure 5.2: Scenario generation template form for a fast-food restaurant system.

fast-food restaurant system, and an Automatic Banking Machine (ABM) system. The texts of the structured scenarios are parsed using this domain model and the resulting instances of classes in the domain model are stored in the objectbase. The schema of the objectbase has an entry for each of the classes in the domain model as well as an index entry as its primary key. A scenario (as a record) in the populated objectbase contains: instances of the different classes resulting from parsing the scenario, and; a unique index representing the scenario.

As shown is Figure 5.3, in our model every instance of the *Scenario* class is composed of one or more instances of *Actor*, *Working information*, and *Action* classes, and zero or more instances of *Dependency* class. Moreover, every *Scenario* instance is associated with one or more instances of *Goal* class. In the rest of this section the classes of the proposed scenario domain model are introduced along with examples from a restaurant system.

Goal: A definition of goal is presented in [12]. In this paper we define goal as follows:

A goal can be *functional*, that is, it corresponds to performing a task; or a goal can be *objective*, that is, it refers to achievement of a quality for the system.

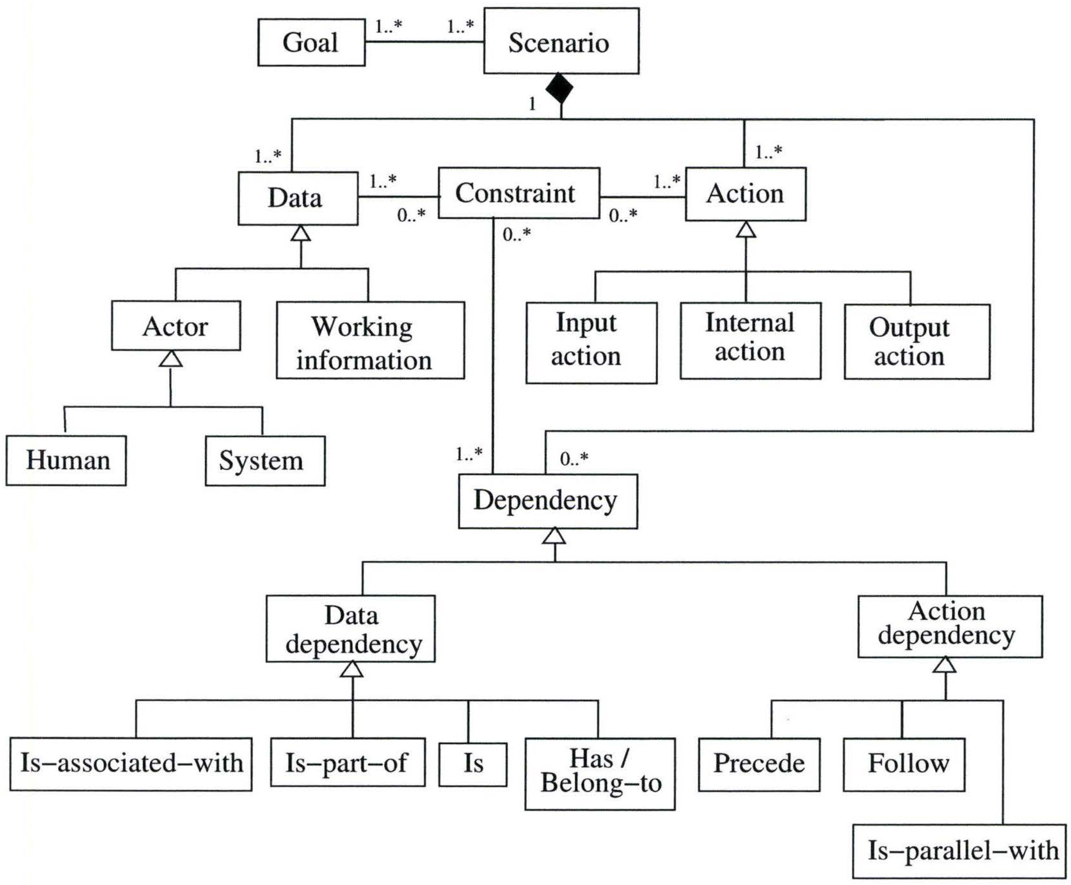


Figure 5.3: Scenario Domain Model to parse a scenario and populate an objectbase.

In general, goals represent the reasons and the desired effects for which the subject system has been produced and used. Examples of goals in a fast-food restaurant system are as follows: handling payment (functional), preparing food (functional), and shortening order preparation time (objective).

Actor: An actor is a "human" or a "system" or a "component of a system" that interacts with other actors during the execution of the scenarios. Examples of actors in a restaurant system include: order taker (human), raw material supplier (system), or food assembly station (component of a system).

Action: An action is an activity that is performed by an actor during the execution of the scenarios. Generally, an action manipulates an instance of *Working information* - which will be explained shortly. Actions can be categorized into three different types, *Input, Internal,* and *Output*, based on the scope of their working information manipulation. Examples include: taking order (input), computing the price of an order (internal), and delivering food (output).

Working information: Working information refers to the information that is manipulated (exchanged, transported, communicated, operated on, stored in the system, etc.) by the scenario's actor during the execution of the scenario. Examples are: customer's order, raw material, menu item, and item price.

Dependency: A dependency refers to a binary relation between two instances of the classes *Actor, Action,* and *Working information.* During parsing of a scenario, dependencies are established both between the newly generated instances of domain model classes (corresponding to the current scenario), and also between these newly generated instances and the previously stored instances in the objectbase.

In our domain model, a dependency can be of type *Data dependency* or *Action dependency.* Data dependency can be one of the following subtypes: *Is*, e.g., "order taker *Is* an employee"; *Is-associated-with*, e.g., "every menu item *Is-associated-with* a recipe", or "every kitchen-table *Is-associated-with* many order-items"; *Has*, e.g., "every menu item *Has* a name"; *Belong-to*, that is the inverse[1] of *Has*, e.g., "an ID *Belongs-to* an employee"; *Is-part-of*, e.g., "a kitchen *Is-part-of* a restaurant". The multiplicity of the participants in a dependency should

[1]For some dependencies, their inverse dependencies are also included in the domain model to facilitate back tracing of dependencies in generating design from objectbase.

be mentioned in the dependency instance.

Action dependency can be one of the following subtypes: *Precede*, e.g., "order payment *Precedes* order delivery"; *Follow*, that is the inverse of *Precede*, e.g., "order preparation *Follows* order taking". *Is-parallel-with*, e.g., "sending order to assembly station *Is-parallel-with* sending order to preparation station".

The proposed scenario domain model in Figure 5.3 includes a *Constraint* class with association relations with *Data* and *Action* classes. This class contains information about the possible constraints that may be associated with instances of each subclass of *Data*, *Action*, and *Dependency*. Examples of these constraints include: *capacity*, *value range*, *ordinal*, *timing*, *privilege*, etc. As an example, a restaurant system may have "*younger than 10*" as a *constraint* associated with an *actor* of some scenario, in order to perform a specific *action* such as "offering kids deal".

5.5 Design diagram generation(stage 3)

In this section, we discuss the guidelines for generating the software system's design diagrams corresponding to data, function, and network aspects of the system using the information stored in the objectbase. In order to illustrate our approach we use some common design diagrams.

- **Data diagram**. Entity-relationship (ER) and class diagrams are appropriate models to represent the data aspect of a system. The following guidelines specify the generation of ER and class diagrams from the information in the objectbase.

  i) Instances of *Actor* and *Working information* are candidates for entities (in ER diagram) or classes (in class diagram) and their corresponding attributes.

ii) Instances of *Is* class are used to find generalization and inheritance relationships in class diagrams, i.e., A *Is* B, means A is subclass of B, or B is superclass of A; where they imply relationships in ER diagrams.

iii) Instances of *Is-associated-with* class are used to find association relationships.

iv) Instances of *Has* and *Belong-to* classes are used to find the attributes of the entities or classes, i.e., A *Has* B (or B *Belongs-to* A) means B is an attribute of entity (class) A.

v) Instances of *Is-part-of* class imply decomposition relationships in class diagrams, where they imply relationships in ER diagrams.

- **Function diagram.** The function of a system is well represented by function diagram or activity diagram. The following guidelines specify the generation of these diagrams.

  i) Functions (in function diagram) and activities (in activity diagram) are instances of the subclasses of *Action* class in the scenario domain model.

  ii) The time-order of actions (functions or activities) are determined by instances of the *Follow* and *Precede* classes.

  iii) The participants of a *Is-parallel-with* dependency are performed concurrently.

  iv) The conditions under which an action can be performed is determined by instances of the *Constraint* class in the related scenario.

  A detailed procedure for constructing the function view is presented using a case study in Section 5.6.3.

- **Network diagram.** The network of a system is usually modelled by diagrams consisting of nodes and interconnections, where a node represents a system, a component of the system, or a physical unit; and an interconnection represents a communication link between two nodes [56]. The following guidelines generate the network diagram from the objectbase.

i) Instances of subclasses of *Data* in the domain model (typically the class *Actor*) that are in the form of system units are candidates for the network nodes.

ii) Association relations between the nodes that imply a business connection represent the network interconnections between the nodes.

The realization of the scenario to design transformation will be presented as a case study in the next section.

5.6 Case study: Fast-food restaurant system

In this section, we follow the defined steps within the proposed framework in order to generate, validate, and transform a set of scenarios into software design diagrams. To avoid the complexity of design construction for the whole system at once, we identify different groups of high-level system functionality as candidate components to be designed. In this context, we apply an incremental transformation process by constructing the design diagrams for individual system components. In the case of a typical restaurant system, the identified components include: *order taking, assembly, preparation, inventory,* and *management.* We focus on the *order taking* component of the restaurant system and in the rest of this section we discuss the three stages of the scenario to design transformation framework discussed in Section 5.2.

5.6.1 Stage 1: Scenario generation

We assume that the following scenario has been defined during the requirement elicitation phase:

"Compute and report the total amount due for the order and sending the completed order for delivery to the assembly station."[2]

At this step, we apply the syntactical conformance test on the scenario and it turns out that the scenario can instantiate the subclasses of *Goal, Action* and *Working information* classes in the domain model, however, the scenario lacks information about the *Actor* class. Since the scenario failed the test we adjust the scenario and the new version would be:

"Order taking station (OT) computes and reports the total amount due for customer orders and sends the paid orders to the assembly station."

However, this scenario corresponds to more than one business-rule action. It contains both *computation of the amount due of an order* and *sending the order to the assembly station.* Therefore, we break the above scenario into two finer scenarios each referring to a single business rule, as follows:

- Scenario #1: *"Order taking station computes and reports the price of the orders."*
- Scenario #2: *"Order taking station sends the paid orders to assembly station."*

[2]To accelerate customer service, the paid orders are directly sent to the assembly station to be assembled.

These two structured scenarios are then added to the set of qualified scenarios Other scenarios that are generated for this component are as follows:

- Scenario #3: *"Order taker logs into the OT station using ID and password."*,
- Scenario #4: *"Order taker initiates orders."*,
- Scenario #5: *"Order taker adds and removes (edit) menu items of an unpaid order."*,
- Scenario #6: *"Order taker enters the amount of money received from the customer (cash-in) to OT station."*
- Scenario #7: *"Order taker defers the payment of orders."*
- Scenario #8: *"Order taker reviews the orders."*
- Scenario #9: *"Order taker calls-back unpaid orders."*
- Scenario #10: *"Order taker returns the change (and receipt) for the order."*
- Scenario #11: *"Order taker sends the cash exceeding cash limit to the cash safe."*
- Scenario #12: *"Order taker logs out from his/her ID."*

Similarly, these scenarios are adjusted and added to the set of qualified scenarios.

5.6.2 Stage 2: Scenario decomposition

At this stage, each qualified scenario is mapped onto the domain model to instantiate different classes of the domain model and the resulting instances are stored in the objectbase. The mapping for the first two scenarios are as follows:

$Index$	$Actor\|_{System}$	$Actor\|_{Human}$	$Working\ information$	$Action\|_{Input}$	$Action\|_{Internal}$	$Action\|_{Output}$
1	OT Station	-	order,price	-	compute price	report price
2	OT Station, ASM station	-	paid order	-	-	send paid order to ASM station
3	-	order taker,OT station	ID&password	-	login to system	-
4	-	order taker	order	-	initiate order	-
5	-	order taker	menu item,unpaid order	-	add/remove menu item	-
6	-	order taker,OT station	cash-in	enter cash-in	-	-
7	-	order taker	order	-	defer payment	-
8	-	order taker	order	-	review	-
9	-	order taker	unpaid orders	-	call-back	-
10	-	order taker	change/receipt	-	-	return change/receipt
11	cash safe	order taker	cash,cash limit	-	-	send money to cash safe
12	-	order taker	ID	-	log out	-

Index	Is-associated-with	Belong-to	Is-part-of	Follow	Precede
1	-	(price,order)	(report price, compute price)	(report price, compute price)	-
2	-	-	(1,paid order,1,order)	(send paid order to ASM station, report price), ...	-
3	-	(ID&password,order taker)	-	-	(login to system, send paid order to ASM station), ...
4	(1,order taker,n,customer order)	-	-	(initiate order, login to system)	(initiate order, compute price)
5	(n,menu item,1,order)	-	-	(edit order, initiate order), ...	(edit Order, compute price), ...
6	-	(cash-in,order)	-	(enter cash-in, report price), ...	(enter cash-in, send paid order to ASM station), ...
7	-	-	-	(defer payment, edit order), ...	-
8	-	-	-	(review orders, login to system)	-
9	-	-	(1,unpaid order,1,order)	(call-back unpaid orders, login to system)	(call-back unpaid orders, enter cash-in), ..
10	-	(change/recepit,order)	-	(return change/receipt, enter cash-in), ...	(return change/receipt, send paid order to ASM station)
11	(1,cash,1,OT station), (cash safe,OT station)	(cash limit,OT station)	-	(send money to cash safe, send paid order to ASM station)	-
12	-	-	-	(log out ID,login to system),...	-

Scenario #1 decomposition:

$goal = taking\ order\ \&\ handling\ payment$

$actor|_{System} = order\ taking\ station$

$information = order,\ price$

$action|_{Internal} = compute\ price$

$action|_{Output} = report\ price$

$data\ dependency|_{Is\ associated\ with} =$

$(1, OT\ station, n, order)$[3]

$data\ dependency|_{Belong\ to} =$

$(price, order)$

$action\ dependency|_{Precede} =$

$(compute\ price, report\ price)$

Scenario #2 decomposition:

$goal = taking\ order\ \&\ assembling\ order$

$actor|_{System} = order\ taking\ station$

$actor|_{System} = assembly\ station$

$information|_{Internal} = paid\ order$

$action|_{Output} = send\ paid\ order\ to\ assembly\ station$

$data\ dependency|_{Is\ part\ of} =$

$(paid\ order, order)$

$action\ dependency|_{Follow} =$

$(send\ paid\ order\ to\ assmbly, compute\ price)$

[3]The number preceding each item shows its multiplicity in the association, where "n" means "many". These numbers may be determined directly from the scenario, or from domain knowledge. In case of no clue for multiplicity, it can be omitted.

$action\ dependency|_{Follow} =$
$(send\ paid\ order\ to\ assmbly, report\ price)$

As it is shown, in decomposition of Scenarios #1 and #2, different classes of domain model have been instantiated according to the user's interpretation of the relations in the structured scenarios and comparison with the scenario domain model, in the light of domain knowledge. Since Scenario #1 was the first scenario that was mapped onto the domain model and whose instance objects are stored in the objectbase, its instances of *Data dependency* and *Action dependency* classes are only between the instances of this scenario. However, in Scenario #2, dependencies are between both its own and also Scenario #1's instances; e.g., the *follow* dependency between "send order to assembly" action (from Scenario #2) and "compute price" action (from Scenario #1). The same process is repeated for every generated scenario.

Figure 5.4 presents a part of the objectbase that is populated with instances of *Data* and *Action* and five *Dependency* classes by Scenarios #1 to #12.

5.6.3 Stage 3: Design diagram generation

In this stage we follow the procedure presented in Section 5.5 to construct the data, function, and network diagrams.

Data diagrams. According to the instances stored in different *Data* columns (i.e., $Actor|_{System}$, $Actor|_{Human}$, and $Working\ information$) of the objectbase, candidate entities (ER diagram) or classes (class diagram) and attributes are: *order taker, OT station, ASM station, order, menu item, unpaid order, paid order, price, cash, cash safe, change&receipt, cash-in,* and *ID&password.* Similarly, the depen-

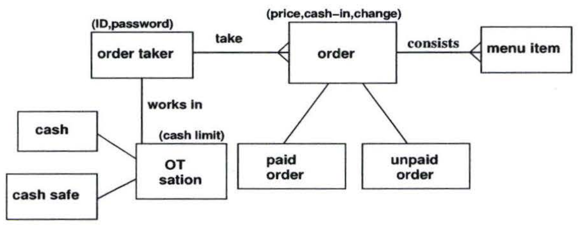


Figure 5.5: Generated Entity-Relationship diagram for the order taking component.

dencies among these candidates are stored in the object base (under *Is*, *Belong-to*, ... columns). Using this information, the constructed ER diagram for order taking component is shown in Figure 5.5.

As an example, we explain how the "order taker" and "order" entities, their attributes, and the dependencies between them are extracted. $Actor|_{Human}$ entry of Scenario #3 (row #3 in Table 1) is the first place that "order taker" is found. Since "order taker" does not appear on the left-hand side of a *Belong-to* dependency under the *Belong-to* column in the objectbase, we conclude that "order taker" is a candidate "entity" and not an "attribute". The same fact is true for the "order" entity where it is first found in Scenario #1 under the *Working information* column. To find the attributes of "order taker" we look for *Belong-to* dependencies (under *Belong-to* column) with "order taker" on the right-hand side. There is one such a dependency in Scenario #3 with "ID&password" on the left-hand side. Therefore, we consider "ID&password" as attributes of "order taker" entity. With the same approach we find "price", "cash-in", and "change/receipt" as attributes of "order" entity. Finally, a dependency between "order taker" and "order" entities is found in Scenario #4 under the *Is-associated-with* column. We can annotate the associations with appropriate names, such as "take" for this association.

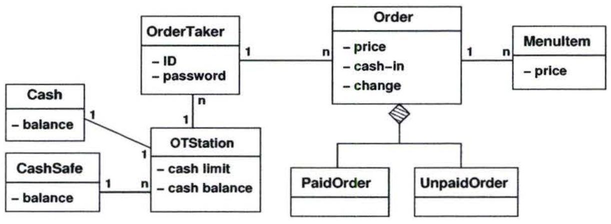


Figure 5.6: Generated class diagram for order taking component.

Function diagrams. By taking the procedure presented for *Function diagram* in Section 5.5 we extract the following actions: *Login to system; Log out the system; Review orders; Initiate new order; Call-back deferred orders; Add/remove menu items to/from orders; Compute total amount due; Report total amount due; Defer payment; Enter Cash-in; Return change-and-receipt; Send order to ASM station; Send cash to cash safe.* The corresponding dependencies between these actions are stored in different *Action dependency* columns of the objectbase. The resulting function diagram for the order taking component constructed from these actions and their corresponding dependencies is shown in Figure 5.7. The process to achieve this diagram from the objectbase is explained below.

For the sake of understandability, after extracting all the actions we may annotate each action name by the related information in the corresponding row of the action in the objectbase. For example, the action "Login to system" in Scenario #3 may be represented as "Login to system using ID&Password" by utilizing the information under the column *Working information.* In order to simplify drawing the function flow diagram, we change every *Precede* dependency with *Follow* dependency and switch the place of corresponding actions in the dependency. In this setting, "*action1 precedes action2*" is replaced by "*action2 follows action1*". Table 2 presents the list of all actions and their "*following*" actions for the order taking component,

Table 5.1: List of actions in order taking component and corresponding to *Follow* relation.

Index	Action	Follows$^+$
1	Login using ID & password	-
2	Logout the system	1
3	Review orders	1
4	Initiate order	1
5	Call-back unpaid orders	1
6	Edit orders	1,5
7	Compute price	1,5,6
8	Report price	1,5,6,7
9	Defer order payment	1,5,6,7,8
10	Enter cash-in	1,4,5,6,7,8
11	Return change & receipt	1,4,5,6,7,8,10
12	Send order to assembly station	1,5,6,7,8,10,11
13	Send excess cash to cash safe	1,4,5,6,7,8,10,11,12

where $Follows^+$ denotes the transitive closure of the relation $Follows$. Therefore, if $action2$ $Follows^+$ $action1$ then there must be a sequence of one or more arrows that connect *action1* to *action2*.

Figure 5.7 illustrates the function diagram for the order taking component that is constructed using Table 2 and according to the following guidelines:

Step 1. Sort the actions based on the number of actions they follow in an ascending order and then start drawing from the beginning of the sorted action list. Table 2 illustrates such a sorting.

Step 2. When there are two or more actions (e.g., B, C, ...) that *immediately*

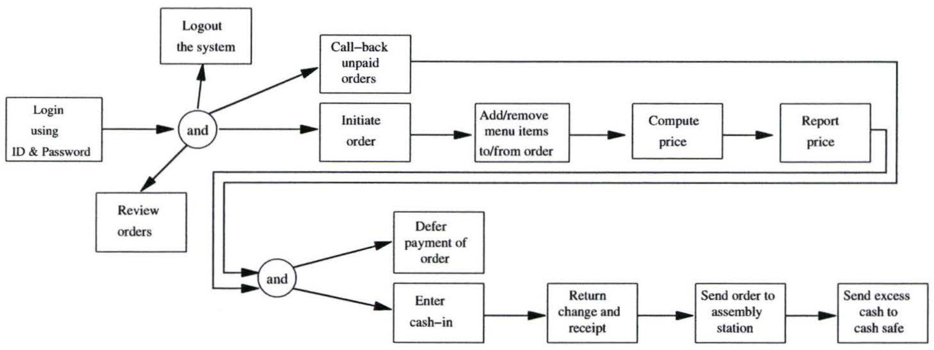

Logout the system
Login using ID & Password
and
Review orders
Call–back unpaid orders
Initiate order
Add/remove menu items to/from order
Compute price
Report price
and
Defer payment of order
Enter cash–in
Return change and receipt
Send order to assembly station
Send excess cash to cash safe

Figure 5.7: Generated function diagram for order taking component.

follow an action A (in Table 2), a *choice* situation has happened where one of the *follow* actions will be executed based on the result of choice. However if there exist a constraint, say between A and B, then selection of B is also conditioned to the satisfaction of the constraint between A and B. Finally, if there is a *Is-parallel-with* dependency between two actions, say B and C, the selection of B (or C) results in parallel execution of both actions B and C. The *choice* situation is represented by the *OR* bubble in function diagram and *diamond* in activity diagram; and *constraints* can be represented by *guard* attributes. Also, the parallel execution situation is represented by *AND* bubble in function diagram and *fork* in activity diagram.

Step 3. When several actions immediately precede a single action a *join* situation has happened which is shown by proper notations in both diagrams.

Network diagrams. To produce a node and interconnection diagram that covers the network view of the system we take the approach presented for *Network diagram* in Section 5.5. The extracted network nodes are *order taking station, assembly station,* and *cash safe* which are found in the $Actor|_{System}$ and $Actor|_{Human}$ columns of the objectbase; and two extracted network links are between *order taking station* and

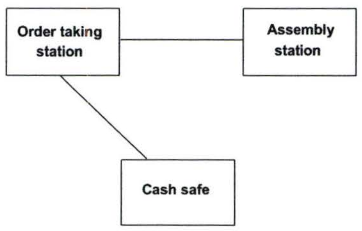


Figure 5.8: Generated network diagram for fast-food restaurant system.

assembly station (a link for order taking to send order to assembly), and between order taking and cash safe (a link to transport cash to cash safe from order taking). The resulting network diagram is shown in Figure 5.8.

Figure 5.6 and Figure 5.9 illustrate the class diagram and activity diagram of the order taking station which are constructed using the procedure discussed earlier in this subsection. Also, Figure 5.10 illustrates the complete class diagram of the restaurant system that is achieved through incrementally constructing the class diagrams for each component and connecting them using the dependencies between their classes, i.e., inter-component dependencies.

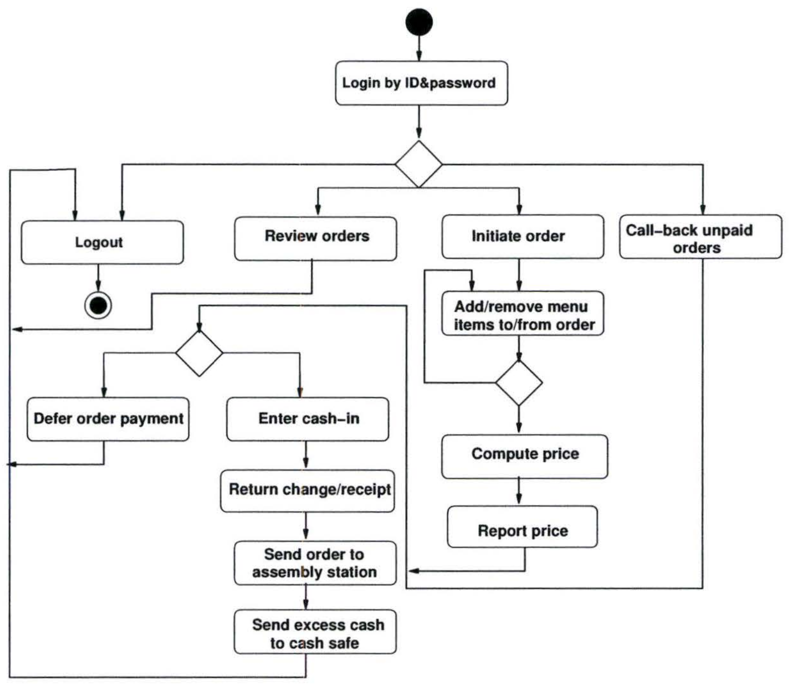


Figure 5.9: Generated activity diagram for order taking component representing.

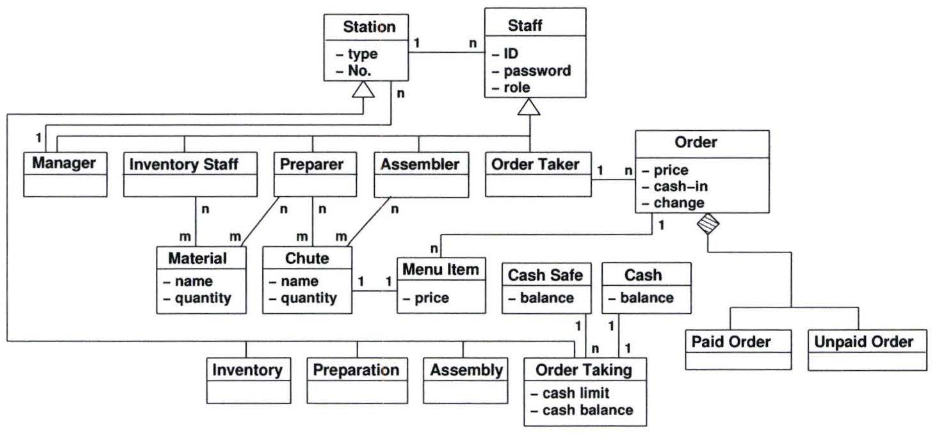


Figure 5.10: Generated class diagram of the whole restaurant system.

Chapter 6

Behavior view and structure view recovery

6.1 Introduction

There is much similarity between the issues arising in architecture recovery of monolotic software systems into logical components and issues of designing distributed applications. The target of such architecture recovery practice is to cluster groups of objects into components in such a way that maximizes the cohesion of each component and minimizes the coupling between different components. Similarly, an objective of designing distributed applications is putting relevant components together to minimize unnecessary inter-component interaction. There exist a large amount of static analysis-based approaches for architecture recovery of systems into a group of components. For example, [41] presents a tool supported user-assisted static analysis based approach to recover the architecture of the software system into cluster of components using data mining techniques. The application of data mining techniques reveals associations between elements of the software system. The recovery process identifies a sub-optimal transformation from a user defined high-level view of the system using architecture query language, namely a conceptual architecture, into a collection of source code components, which is a concrete architecture. A successful match yields a restructured system that conforms with the given pattern architecture constraints.

However, as mentioned earlier, static analysis approaches consider only the "existence" of data/controls dependencies among software components and ignores dynamic information such as frequency of such dependencies being used in the runtime execution of the system. Therefore, providing least possible inter-component interaction, requires dynamic analysis of dependencies between software entities to capture their runtime behavior and finally reflecting the result of dynamic analysis in the component clustering practice.

In this chapter, we present an interactive tool-supported environment for architecture recovery of software systems using their both static and dynamic properties towards a component based architecture. In this regard we enhanced the Alborz [37] architecture recovery tool to accommodate dynamic information in combination with static information. In our approach, the software system is transformed from source code to an attributed relational graph representation. As the dynamic analysis step, execution profiles resulting from execution of a set of scenarios are studied to find frequencies of dependencies being used in the execution. These frequencies are weighted by a proper factor and used in combination with static analysis information to calculate association strength values between software entities in source graph. The recovery process is modelled as a Valued Constraint Satisfaction Problem (VCSP) that matches a high-level conceptual architecture of the system with the source system.

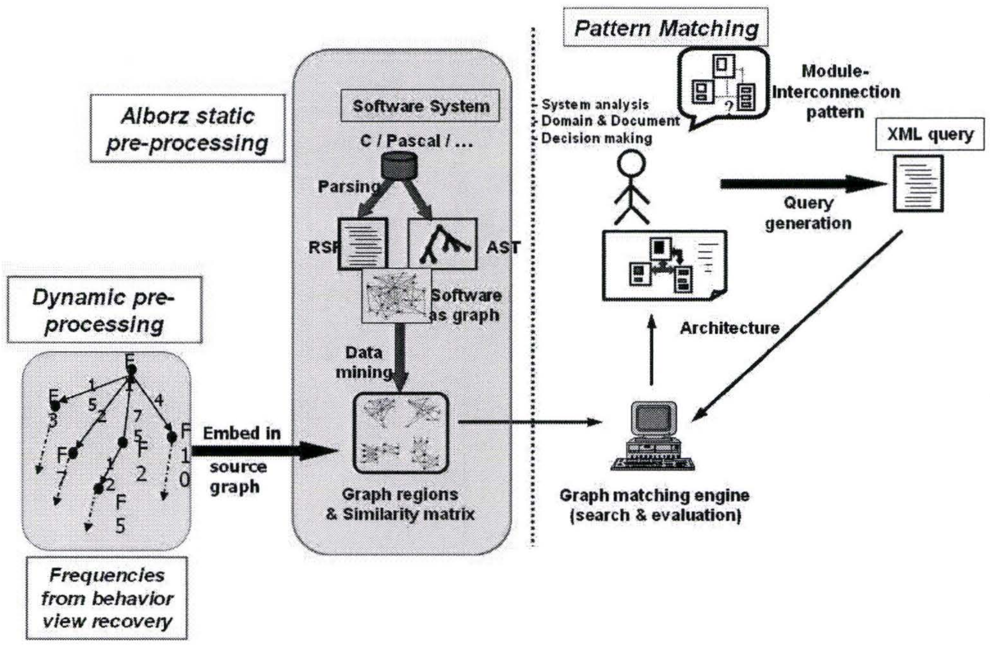


Figure 6.1: Enhanced Alborz architecture recovery environment.

6.2 Combined static-dynamic model

Figure 6.1 illustrates the interactive tool-supported environment for the proposed combined static-dynamic architecture recovery model. In a nutshell, this model combines the static and dynamic information of system to recover the architecture of the system into a collection of distributed components using a pattern matching technique modelled as a Valued Constraint Satisfaction Problem. As shown in Figure 6.1, the architecture recovery occurs within three major stages as follows:

Stage 1 (Static pre-processing): In the first stage of this framework, the software system is parsed into *abstract syntax tree* (AST) using a source-code parser and then transformed into a higher level of abstraction representation, called the *source graph* based on a pre-defined domain model. In our approach, we use the attributed relational graph notion defined in [18] for the source graph. In this graph notation, nodes represent software constructs, such as: *functions* and *variables*, and edges represent relationships between constructs, such as *function-call* and *variable-use*. The nodes and edges comply with a specific domain model, namely an *abstract domain model*. In this approach, we use the domain model presented in [41]. Such a domain model provides programming language independence for the recovery process. Furthermore, common attributes that are inherited by every entity (or relation) are defined in the abstract domain model. Consequently, in the software representation stage, an association-based similarity matrix that contains the mutual similarity of all the entities of the system is generated. This stage will be further discussed in Section 6.3.

Stage 2 (Dynamic pre-processing): In the dynamic pre-processing stage, first the software system is instrumented by a dynamic profiler tool, such as "gprof". Then, a set of task scenarios that may cover the whole or a part[1] of the features of the system are executed on the system one at a time, and the resulting execution call graph profiles are captured by the dynamic profiler. By analyzing the execution profiles, the functions that were invoked as well as the number of times that each function-call is performed is found. The found functions that are invoked during the execution of set of scenario represent a core for the implementation of the features present in the scope of scenarios. On the other hand, the frequencies of travelled function-calls are used to infer a weight value for each function-call. This value is reflected in the source graph by embedding it as an attribute in its corresponding edge in the source graph. This attribute is used as the representor of the dynamic analysis in cost evaluations during the interactive pattern matching stage. Also, the

[1]The scenarios set can cover a subset of system features, in which case the dynamic analysis would be focused on selected features.

dynamic weight is used in the process of determining the domain for each node of the graph. The dynamic pre-processing stage will be discussed in more details in Section 6.4.

Stage 3 (Interactive pattern matching): In this stage, the user defines a conceptual architectural pattern of the system components (subsystems) and their interactions based on: domain knowledge, system documents, or tool-provided system analysis information. In an iterative recovery process, the user constraints the architectural pattern and the tool provides a decomposition of the system entities into components that satisfy the constraints using valued constraint satisfaction problem techniques. In this approach, the architectural pattern is viewed as a collection of potential components and interconnections, where each component represents a group of placeholders for the system entities (i.e., functions, types, variables) to be instantiated, and each bundle of interconnections (one relationship) between two components represents data/control dependencies between two groups of placeholders in two components. The minimum/maximum sizes and the types of both placeholders and the interconnections are considered as free parameters to be decided by the user (respecting the allowed relation between two entities). This yet un-instantiated module-interconnection representation (can be referred to as conceptual architecture) is directly defined for the tool, using a standard markup language such as XML. The entire recovery process is modelled as a Valued Constraint Satisfaction Problem (VCSP). Interactive pattern matching stage is and will be further discussed in Section 6.5.

6.3 Static pre-processing

In this section we present a brief summary of the work presented in [39] regarding software system representation for architecture recovery and further analysis. In the proposed techniques in this thesis, we use their proposed approach for representing the software system. However, their work is based on static analysis of system only, while we combine the static and dynamic analysis. As it is going to be discussed in later sections, we extract dynamic information from the system by running a specific set of scenarios and then we embed the dynamic information in the structure of the system to enhance the architecture recovery process. To do so, we also make some enhancement to the representation model presented in the referenced approach, as it is going to be discussed later in this section.

For the purpose of our software analysis technique, in the static pre-processing stage we transform the source-code representation of the system to a higher level of abstraction, such as an attributed graph representation. This transformation is done because the source-code is too detailed to perform a meaningful architecture recovery practice. However, even the graph representation of the whole of a medium-size system is too large to be tractable for the architecture recovery process. Therefore, the graph is also partitioned into some meaningful smaller subgraphs, that are analyzed in an incremental way. The static pre-processing stage consist of different activities as follows: fact extraction, source graph generation, search space reduction, and association-based similarity matrix generation. In the following each of these activities are discussed.

6.3.1 Fact extraction

In the fact extraction step, the software system is parsed to generate an *abstract syntax tree* (AST). An AST, which can be represented as a set of entity-relationship tuples, contains all the constructs corresponding to the programming language of the software system, e.g., C. In this approach, we used the Refine parser [34] as our fact extractor.

6.3.2 Source graph generation

As was mentioned earlier, to pursue a tractable architecture recovery practice, the software system should be transformed from source-code (or AST) into a higher-level abstract representation. In this regard, attributed-graph representation of a software system has been used in different software analysis approaches [40, 41, 28, 19]. Similarly in this approach, we use a graph notation, called the *source graph* to represent the system under study. To do so, a domain model should be defined that specifies the types of nodes, edges, and attributes in the graph. In this work we use the same domain model namely *abstract domain model* defined in [41] that specifies architectural level entities for recovery of software modules, such as: file (File-abs), function (Function-abs), aggregate types (Type-abs), and global variables (Variable-abs) [2] and their relationships, such as: file-containment, function-call, type-use, and variable-use. This domain model supports both file-level and function-level architecture recovery. However, the focus of this work is on function-level analysis only.

The attributed-graph representation of the source graph is defined as a six-tuple $G^s = (N^s, R^s, A^s, E^s, \omega^s, \psi^s)$ as follows:

[2]The focus of this work is on the function-level analysis.

- $N^s : \{n_1, n_2, ..., n_n\}$ is the set of nodes defined in the domain model
- $R^s : \{r_1, r_2, ..., r_m\}$ is the set of edges defined in the domain model
- A^s : alphabet for node attributes and their values
- E^s : alphabet for edge attributes and their values
- $\omega^s : N^s \rightarrow (A^s \times A^s)^p$: a function that returns "node attribute, node attribute value" pairs, where p denotes the number of node attributes
- $\psi^s : R^s \rightarrow (E^s \times E^s)^q$: a function that returns "edge attribute, edge attribute value" pairs, where q denotes the number of edge attributes

Since the proposed architecture recovery in this research is at the function level, the nodes of the source graph are of types: *function, variable,* and *type*; and the types of the edges are: *function-call*, *variable-use*, and *type-use*. Moreover, there are a number of attributes defined for nodes and edges in the abstract domain model in [41]. The most important attributes for nodes are: *name*, which refers to the name of the node in the source code; *type*, which is one of the node types mentioned above; and *id*, which is a unique identifier for the node. The most important attributes for the edges are as follows: *from* which indicates the source of the edge; *to*, which indicates the sink of the edge; id, a unique identifier; and *type*, which is one of the edge types mentioned earlier. Figure 6.2 illustrates an example of a very small source graph with 9 nodes and 16 edges. The following is an example of applying ω^s and ψ^s functions on a node and an edge of this graph:

$\omega^s(n_7)$ = ((name, "./calculateAverage"), (type, function), (id, F7)),

$\psi^s(r_9)$ = ((from, n_7), (to, n_6), (type, function-call)).

In addition to the aforementioned attributes, we add a *frequency* (or *freq* in short) attribute to the edges of the source graph. This attribute serves as the representative

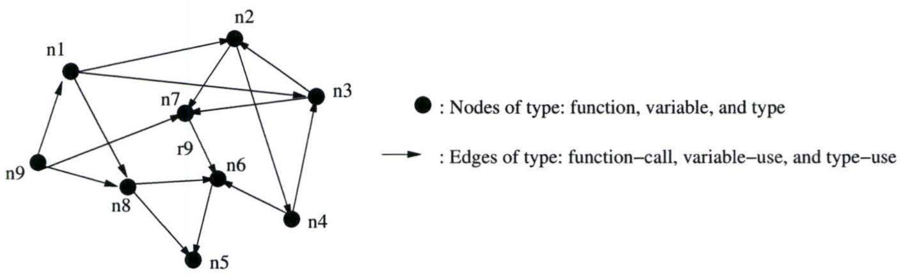

n1
n2
n3
n4
n5
n6
n7
n8
n9
r9
● : Nodes of type: function, variable, and type
⟶ : Edges of type: function–call, variable–use, and type–use

Figure 6.2: An example of a small source graph with 9 nodes and 16 edges.

of the dynamic analysis step in the source graph. The *frequency* attribute of an edge (typically an edge of type function-call) reflects the number of times the edge has been traversed in a specific execution of the system. More details about this attribute will be presented in the discussion of the dynamic pre-processing stage.

In the rest of this thesis, without the loss of generality, we refer to the source graph as a tuple of $G^s = (N^s, E^s)$.

6.3.3 Similarity matrix generation

In a further step, in the software representation stage, a similarity metric is defined between every pair of system entities based on *maximal association*. Maximal association refers to a maximal set of entities that all share a similar relation with another maximal set of entities. This property is a key to grouping entities into cohesive modules in terms of specific relations. Data mining techniques are widely used for extracting maximal association. The Apriori data mining algorithm [7] for example, is used in the Alborz tool [37] that supports our recovery process. The notations *basket* and *item* in the data mining domain, refer to "entity" (such as "function F2") and a pair of "relation and entity" (such as "Use-F function F2") in the reverse engineering domain, respectively. Data mining *association rules* (such as "25% of the baskets that contain item X, also contain item Y") can be extracted using *frequent itemsets*, where frequent item sets can be found using the Apriori algorithm. Generally, a k-frequent itemset is a set of k items where all the items are contained in every basket of a group of baskets [3]. Consequently, the generated frequent itemsets are sorted decreasingly based on their cardinality and stored in a database. A *similarity* measure is defined between each two entity in a way that two entities that are alike posses a higher similarity value than two entities that are not alike. In this approach, association values between nodes of the source graph, are determined using the notion of *associated group*. An associated group of graph nodes forms when two or more source nodes (nodes that edges originate from them) share one or more sink nodes (nodes that edges point to them). By referring to the source nodes as basketset, and sink nodes as itemset, the entity-similarity association between two nodes e_i and e_j (shown as: $entAssoc(e_i, ej)$) belonging to all associated groups g_x is defined as:

$$entAssoc(e_i, ej) = max(|itemset(g_x)| + \; w \times |basketset(g_x)|) \tag{6.1}$$

where $|itemset(g_x)|$ is the cardinality of shared entities, $|basketset(g_x)|$ is the cardinality of sharing entities, and $0 < w < 1$ is the weight of sharing entities. Since e_i and e_j may belong to more than one associated group g_x, as shown in Formula 6.1, the maximum value of their associations in all the associated groups is considered as their association value. The more the weight w is close to zero, the less the entity association is dependent on the number of sharing items. Based on some empirical result we use a value of $w = 0.5$. Figure 6.3 illustrates the calculation of entity similarity association in a sample source graph.

[3]Frequent-itemset extraction is discussed in more detail in [39].

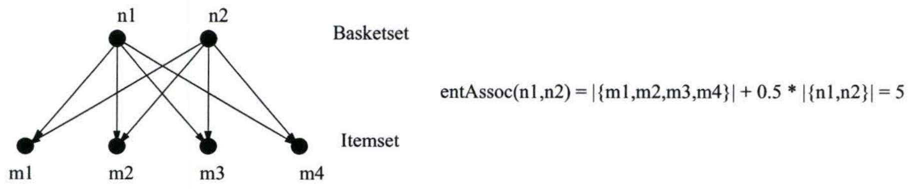


Figure 6.3: An example of entity similarity association calculation with $w = 0.5$.

6.3.4 Search space reduction

Considering the number of entities in a medium size software system (usually more than 1000 entities), searching the whole search space (source graph) in the pattern matching stage is an intractable problem. Hence, we must restrict the search domain for each module to a group of eligible entities. To do so, we semantically decompose the source graph into smaller regions, called *source regions* where each source region consists of a number of entities that are associated with an entity in that region, namely a *main-seed*. More formally, a source region $G_j^{sr} = (N_j^{sr}, R_j^{sr})$ is a subgraph of the source graph, i.e., $N_j^{sr} \subseteq N_s$ and $R_j^{sr} \subseteq R_s$, that is related to a node n_j such that each node $n_i \neq n_j$ in G_j^{sr} satisfies the property $entAssoc(n_j, n_i) > 0$. The node n_j is called the main-seed of G_j^{sr}. There are as many source regions as there are nodes in the source graph and they can be found using Apriori data mining algorithm. Using the notion of source region, we define *domain* D^{n_j} of a node n_j as the set of 3-tuples (n_d, s_d, f_d) where:

$n_d \in N_j^{sr}$, and

$s_d = entAssoc(n_j, n_d)$, and

$f_d = edgeFreq(n_j, n_i)$ is the frequency of the edge connecting n_j and n_d which is going to be discussed in next section.

Consequently, using the domains we restrict the search space for each module to the domains of its main-seeds.

Moreover, since even in a medium-size software system the recovery of all the modules at once is intractable, due to the extremely large amount of generated data, to address the tractability of the matching process, the whole process is divided into k partial matching phases, where k is the number of modules (components) to be recovered.

In the following section, the dynamic pre-processing step of the multi-view framework is discussed.

6.4 Dynamic pre-processing

The dynamic pre-processing stages extracts runtime information from execution profiles resulting from execution of a set of scenarios on software system. The dynamic information that we extract from each system execution are as follows: i) the executed functions along with their frequencies; and ii) the function-calls performed by each function along with their frequencies. To capture this information the following three steps should be taken: 1- instrumentation of the software system; 2- execution of a set of scenarios on the instrumented system; and 3- analysis of the resulting execution profiles. In the rest of this section, each of these steps are discussed after some terminology definitions.

Terminology

- *instrumentation:* 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.

- *feature:* a feature is a realized functional requirement (the term feature is intentionally defined weakly because its exact meaning depends on the specific context). Generally, the term feature also subsumes non-functional requirements. However, in the context of this paper only functional features are relevant, i.e., we consider a feature an observable result of value to a user [16].

- *frequent feature:* a frequent feature is a feature that is used by users of the system more often than other features.

- *scenario:* a scenario is a sequence of user inputs triggering actions of a system that yields an observable result to an actor [9]. A scenario is said to execute a feature if the observable result is executed by the scenario's actions. A scenario may execute multiple features. Scenarios resemble use cases but do not include options or choices, so a use case subsumes multiple scenarios [16]. In this paper, we consider scenarios as sequences of features.

- *subprogram:* a subprogram is a function or procedure according to the programming language. Subprograms are the lowest level kind of components.

- *component:* a component is a group of subprograms along with their related variables and data types that implement a computational unit of a system. Components have import and export relations with other components. The component that calls a function, or uses a variable or data-type of another component is called the exporter and the other component is called the importer component.

- *execution summary:* an execution summary of a given program run lists all subprograms called during the run.

6.4.1 Software system instrumentation

We use the GNU profiler, *gprof* [5], for the purpose of instrumenting the software system and capturing 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 shows the total amount of time your program spent executing each function. The call graph shows how much time was spent in each function and its children It also shows the number of times a function called its children and was called by its parents. In this paper, we use the call graph output of the profiler.

6.4.2 Execution of scenarios

A domain expert is a key actor in designing the scenarios. These scenarios must cover a set of *frequent features* that are of users interest. This is mainly because we are more interested to capture the most-frequent behavior of the system. Therefore, he identifies the frequent features using domain knowledge with guidance of design diagrams (specifically activity diagrams) and then based on the scenario set that was generated in the design view recovery step, he generates a new set of scenarios. The expert is also responsible for eliminating redundancy from the set of scenarios in terms of the covered features, so that the resulting profiles would not be relatively larger than what they should be. After scenario selection, the set of scenarios are executed on the instrumented system.

6.4.3 Analysis of execution profiles

In this step we take a maximum-approximation approach to assign a frequency to each function and function call that is present in the resulting execution profiles. In this approach, we consider the greatest frequency found for each function and function call among the execution profile of all the scenario as the final frequency of that function or function call. Frequencies of function-calls are then embedded in the source graph as the frequency attributes on edges between functions. This attribute is then used in the cost calculations for the Valued Constraint Satisfaction Problem (VCSP) in the next stage.

The default value for frequency attribute for edges in the source graph is *0*. This value will be updated to *freq*$\geq$ *0* after the dynamic analysis (pre-processing) stage, where *freq=0* means the absence of an edge in the runtime execution of system and *freq>0* is the frequency of appearance of that edge in the execution of the system.

6.5 Pattern matching

In the pattern matching stage we perform a supervised valued constraint satisfaction pattern matching technique that incrementally generates software components as cohesive modules of entities, i.e., functions, variables, and data types, that are interconnected through function-call, varibale/type-use or generally imports and export relations. Recovered modules conform with a set of architectural constraints specified by XML language. Each module consists of one or more main-seeds as the core functions of the module and a sub-optimal version of a branch and bound search algorithm is used to collect the group of functions, variables, and data types that are highly statically and dynamically associated to the main-seeds into the module. As it was mentioned in Section 6.3, the search space for a module is restricted to the entities in the search domain(s) of the corresponding main-seed(s). The presence or absence of a node in a module is determined by the set of constraints and the cost function of the VCSP framework that takes into account both static and dynamic information.

In the rest of this section we discuss the different steps of the pattern matching stage. We start by briefly explaining the notion of the Valued Constraint Satisfaction Problem (VCSP). Then we discuss our modelling of the architecture recovery process based on VCSP. Finally, we explain the iterative pattern matching step using a branch and bound search algorithm [42].

6.5.1 Valued Constraint Satisfaction Problem

The Valued Constraint Satisfaction Problem (VCSP) framework [44] is an extension of the conventional Constraint Satisfaction Problem framework (CSP), that allows for dealing with over-constrainted problems [43]. In the VCSP framework, a valuation (or cost) is associated with each constraint. The task of assigning a value to variables in the problem is called an assignment. The valuation of an assignment is defined as the aggregation of the valuations of the constraints which are violated by this assignment. The goal of a VCSP is to find a complete assignment of minimum valuation. Typically, a search algorithm is used to find the 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 a set of associated domains, C a set of constraints between the variables, and f a valuation (cost) function.

6.5.2 Modelling the recovery process

As mentioned earlier, in our approach the recovery is modelled as a valued constraint satisfaction problem. The specifications of the problem is presented as an architectural pattern query using XML notation. The XML query (which is going to be discussed in detail in the next section) contains the conceptual components that are going to be recovered. Each conceptual component consists of a number of variables (considered as variables of VCSP), a set of main-seeds assigned, and a a set of link constraints between itself and the concrete ones (i.e. the components that have been already recovered). The links between the components can be of different relation types, such as: function-call, variable-use, type-use. Such relations are generally called import/export relations. The nodes of the source graph, i.e., the functions, variables, and types are considered as the candidate values to be assigned to variables of the VCSP. The domain of each variable is the same as the search domain determined for each corresponding node in the static pre-processing stage of our framework. The valuation or cost function is defined based on the cost of existence of an edge between two nodes. The existence of edges is determined by the values assigned to the placeholders (variables) of the component. The cost is calculated using both static and dynamic information of the system. In this approach we use the branch and bound search algorithm to find a minimum cost valuation of the assignment of the entities in the source model (domain of the variables) into the placeholders (variables) of the query. In our approach, the recovery is performed in an iterative process which recovers one component at a time.

6.5.3 Modelling constraints with XML

Pattern-based architecture recovery techniques provide a high-level conceptual model for the architecture called the architectural pattern [26, 21]. As mentioned earlier in this section, we use XML markup language to model the conceptual architecture or the constraint query. The schema of the XML model we defined for constraint query and a sample query are illustrated in Figure 6.4 and Figure 6.5 respectively. The notation we used to define the schema of queries is the XML Schema language, or XML Schema Definition (XSD), provided by W3Schools [1]. In the following the main elements of the schema are explained.

<query>: this is a "complex" element and contains all the information necessary for the query within its child elements. Each <query> element contains one <name> and one-to-many <contains> elements.

<name>: this element is of type "string" and is an arbitrary name for the query used for later reference.

<contains>: this element is of type "component" and declares a conceptual component that the query aims to recover. Each element of type "component" contains one <componentName>, one-to-many <mainSeed>, one <minSize>, one <maxSize>, and one-to-many <linkConstraint> elements.

<componentName>: this element is of type "string" and declares a unique name for the new component that is going to be recovered.

<mainSeed>: this element is of type "entity" and specifies a main-seed for the component to be recovered. There can be one or more of this element defined for each component. Each element of type "entity" contains <entityName> and <id> elements.

<**entityName**>: this element is of type "string" and specifies the name of the main-seed in the source code.

<**id**>: this element is of type "string" and specifies the assigned unique identifer of the main-seed entity in the recovery process.

<**minSize**>: this element is of type "integer" and specifies the minimum size of the component to be recovered in terms of number of entities in the component, which is restricted to be equal or greater than one.

<**maxSize**>: this element is of type "integer" and specifies the maximum size of the component to be recovered in terms of number of entities in the component, which is restricted to be equal or greater than one.

<**linkConstraint**>: this element is "complex" and it specifies the link constraint between the component under recovery and the already recovered components. Each <linkConstraint> element contains one <linkCompName> and one <upperBound> elements.

<**linkCompName**>: this element is of type "string" and specifies the name of an already existing component between which the component under recovery has a link constraint.

<**upperBound**> this element is a numeration of type "string" and specifies an upper bound for the link constraint between the component under recovery and the component mentioned in <linkCompName>. Possible values for this element are *none, low, average, high,* and *unbounded* that will be discussed later in this section.

6.5.4 Iterative pattern matching

After modelling the conceptual architecture as XML pattern queries, the tool searches for a solution (concrete components) for the pattern query (conceptual components) in an iterative pattern matching stage. As mentioned earlier, the pattern matching is modelled as a valued constraint satisfaction problem. In this regard, a solution is a complete value assignment to all the variables of the problem that has the minimum cost. In this section, we describe the matching of a simple pattern query q of two modules $M1$ and $M2$, and its associated VCSP model. Before we proceed, we define two notions of *similarity (internal) constraint* and *link (external) constraint*:

i) *similarity (internal) constraint:* the static similarity between each pair of the assigned values in a module is determined by considering the shared features of those two values as mentioned earlier in this chapter. 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 to aggregate the static distance values (1 - similarity value) between the candidate value and the values of the already instantiated variables in that module, as a measure of ranking the module by the branch and bound algorithm Since searching the entire search tree is usually not practical for medium-size systems, an upper-bound is defined for the violation of the similarity constraint, so that if the aggregated cost of the violated constraints exceeds the 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.

Other than the similarity mentioned above (static similarity), a dynamic similarity is defined between values (entities) of type function. Dynamic similarity between each pair of the assigned functions is determined by the dynamic frequency of the function-call edge connecting the pair. The dynamic similarity has an inverse relation with the valuation cost of selecting variables ($\frac{1}{freq}$). Therefore combined with the valuation cost of static similarity the valuation function will aggregate to combined static-dynamic distance ($\frac{1-similarityvalue}{1+freq}$) between the candidate values and the values of the already instantiated variables in that module.

iv) *link (external) constraint:* link constraint is defined on the edges between the candidates for the placeholders of the current component with the already recovered components. The constraint is defined by the user in the XML pattern query as an upper-bound for the average of accumulated link cost of the assigned edges that connect the two components. In this context, insertion of an edge e has a link cost $edgeLinkCost(e)$ that is accumulated with the link cost of previously inserted edges for that component. The cost of inserting an edge $e = (n_1, n_2)$, is calculated as follows:

$$edgeLinkCost(e) = 1 + \frac{edgeFreq(e)}{totalFreq}$$

where $edgeFreq(e)$ is the dynamic frequency of the edge e and $totalFreq$ is the sum of all the dynamic frequencies of the edge in the system which is discovered in the dynamic pre-processing step. The above formula considers a cost of 1 for the existence of an edge (static) and a cost in range of zero to 1 for the relative frequency of the edge to the total frequency in the system. The total link cost of two components is defined as the average of the link cost of all of the assigned edges e that connect two components:

$$totalLinkCost = \sum_{\forall e} \frac{edgeLinkCost(e)}{n}$$

where n is the number of the connecting edges e.

The total link cost must not exceed an upper bound. Therefore, if insertion of an edge causes link constraint violation (i.e. exceed the upper bound), the edge is discarded. The upper bound can be one of the following:

- *none*: which means that there should not be any edge between the two components ($totalLinkCost = 0$).

- *low:* which means that the interaction of the two components can be up to 25% of the total interaction in the system $(totalLinkCost - 1 < 25\%)$[4].
- *average:* which means that the interaction of the two components can be up to 50% of the total interaction in the system $((totalLinkCost - 1 < 50\%))$.
- *high:* which means that the interaction of the two components can be up to 75% of the total interaction in the system $(totalLinkCost - 1 < 75\%)$
- *unbounded:* which means that the interaction of the two components can be up to 100% of the total interaction in the system $(totalLinkCost - 1 < 100\%)$ (i.e., there is no limits for the interaction of the two components).

In the following the steps of the mapping are explained:

- **Mapping step 1:** For every placeholder in the conceptual component, we assign a variable v_i in the set of variables V of the VCSP. For each variable v_i we 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 specified for the corresponding component in query q.
- **Mapping step 2:** For every pair of variables in V that correspond to the same module (e.g., $M1$), define a constraint of type similarity (internal) constraint in C. If module $M2$ exports/imports a matching placeholder link to module $M1$ (e.g., a link to function $F1$), assign a constraint of type link constraint from every single variable in module $M2$ to $F1$.

We define the valuation function on the basis of our architectural recovery objectives as follows: i) the average similarity value between the group of entities in a module

[4]The minus 1 is for deducting the accumulation of the 1s that were added for the existence of each edge in the *edgeLinkCost* formula

must exceed a threshold which is determined by the overall properties of the software system; ii) in equivalent static similarity situations, the nodes with higher dynamic weight on the edge connecting them have a higher priority to fill the placeholders; iii) all import/export link constraints should be met. 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.

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

- **Search step 1:** the next variable is selected from the current module to be instantiated;
- **Search step 2:** from the domain of this variable the next value (candidate value) is selected to be assigned to the variable;
- **Search step 3:** all similarity constraints and link constraints between the assigned values are evaluated and checked for satisfaction/ violation;
- **Search step 4:** the cost of the assignment is calculated. If the cost is very high (i.e., higher than the upper-bound), the candidate value is discarded, else, the evaluated cost is used as the ranking criterion for the current variable and the value is put in the proper place of the list of all partially assigned values for future assignment and ranking.
- **Search step 5:** the value assignment with the best possible rank (least average similarity cost while not violating the link constraints) of all or the most possible of the variables is the solution for the valued constraint satisfaction problem. The solution is represented as a concrete component that has import/export relations with previously recovered components. The number of the entities in

the component 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.

The resulting assignment is considered as the solution to the conceptual architecture that was defined by the user in the XML query. After this, the user can define another query to recover another component of the system, or he/she can refine his/her query on the same component and run the pattern matching step again.

In the next chapter, the results of applying our multi-view architecture recovery technique on a medium-sized software system is presented.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com" elementFormDefault="qualified">
<xs:element name="query">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="contains" type="component" minOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="component">
    <xs:sequence>
        <xs:element name="componentName" type="xs:string"/>
        <xs:element name="mainSeed" type="entity" minOccurs="1"/>
        <xs:element name="minSize" >
            <xs:simpleType>
                <xs:restriction base="xs:integer">
                    <xs:minInclusive value="1"/>
                </xs:restriction>
             </xs:simpleType>
        </xs:element>
        <xs:element name="maxSize" >
            <xs:simpleType>
                <xs:restriction base="xs:integer">
                    <xs:minInclusive value="1"/>
                </xs:restriction>
             </xs:simpleType>
        </xs:element>
        <xs:element name="linkConstraint">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="linkCompName" type="xs:string"/>
                    <xs:element name="upperBound" >
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="none"/>
                                <xs:enumeration value="low"/>
                                <xs:enumeration value="average"/>
                                <xs:enumeration value="high"/>
                                <xs:enumeration value="unbounded"/>
                            </xs:restriction>
                        </xs:simpleType>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="entity">
    <xs:sequence>
        <xs:element name="entityName" type="xs:string"/>
        <xs:element name="id" type="xs:string" />
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figure 6.4: XML schema of the constraint query.

```
<?xml version="1.0"?>
<query>
   <name>samlpeQuery</name>
   <contains>
      <componentName>newComp</componentName>
      <mainSeed>
         <entityName>{foo1}</entityName>
         <id>F1</id>
      </mainSeed>
      <mainSeed>
         <entityName>{foo2}</entityName>
         <id>F2</id>
      </mainSeed>
      <minSize>5</minSize>
      <maxSize>30</maxSize>
      <linkConstraint>
         <linkCompName>oldComp1</linkCompName>
         <upperBound>average</upperBound>
      </linkConstraint>
      <linkConstraint>
         <linkCompName>oldComp2</linkCompName>
         <upperBound>high</upperBound>
      </linkConstraint>
   </contains>
</query>
```

Figure 6.5: An example of an XML representing aa constraint query.

Chapter 7

Xfig case study

In this chapter, we present the results of applying the proposed multi-view architecture recovery technique on a medium-size open source software system. We use the Alborz reverse engineering toolkit [38] in our experiments to study the architecture of Xfig drawing tool. Xfig 3.2.3d [3] is an open source, medium-size (80 KLOC), C language drawing tool under X Window system. Xfig is used to interactively draw and edit graphical objects (such as lines, circles, and rectangles) through operations such as copy, draw, move, delete, edit, scale, and rotate. In the following, each step of our experiment is described with respect to the multi-view framework in Chapter 4 and the three views in Chapters 5 and 6.

7.1 Design view generation

In this section we demonstrate the results of applying the three steps of design view generation described in Chapter 5, i.e., scenario generation, scenario decomposition, and design diagram generation.

7.1.1 Scenario generation

Figure 7.1 presents the set of scenarios that are generated using domain knowledge and user interface of the drawing part of Xfig tool. These scenarios conform with the scenario structure proposed in Section 5.3. As an example, Scenario #1 in Figure 7.1 conforms with the scenario structure as follows:

User is an Actor (N=1); *draws* is an Action (N=1); *ellipse* and *radius* are Working Information (N=2); and there is no Constraint related to any of them (M=0 for all).

7.1.2 Scenario decomposition

Each generated scenario in previous step is parsed according to the scenario domain model and the collection of the generated instances of the domain model classes are stored in the objectbase. As it is illustrated in Table 7.1 the actor of all scenarios is "user". This is because of the nature of the Xfig tool where scenarios are not performed based on heavy interaction between the actors, as opposed to the case of interactive systems such as a fast-food restaurant or an automated banking machine (ABM).

As an example of scenario decomposition, the generated instances of scenario domain model classes resulting from decomposition of Scenario #1 are presented below:

Scenario #1

$actor = user$

$information = ellipse$

$information = radius$

$action = draw$

#	**Scenario**
1	*"User draws ellipse by radius."*
2	*"User draws ellipse by diameter."*
1	*"User draws circle by radius."*
2	*"User draws circle by diameter."*
3	*"User draws closed spline by control points."*
4	*"User draws spline by control points."*
5	*"User draws closed interpolated spline by control points."*
6	*"User draws interpolated spline by control points."*
7	*"User draws polygon."*
8	*"User draws polyline."*
9	*"User draws rectangle."*
10	*"User draws rounded corner rectangle."*
11	*"User draws regular polygon."*
12	*"User draws arc by three points."*
13	*"User pictures object."*
14	*"User inputs text."*

Figure 7.1: Generated scenarios for drawing part of Xfig.

$$data\ dependency|_{Is\ associated\ with} = (user, ellipse)$$
$$data\ dependency|_{Is\ associated\ with} = (ellipse, radius)$$

The scenarios shown in Figure 7.1 correspond to "drawing" part of Xfig. Similarly, by generating relevant scenarios the same steps can be repeated to cover the "editing" part of the Xfig.

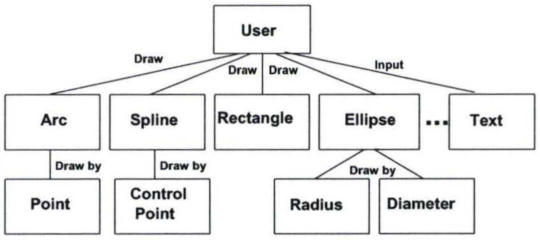


Figure 7.2: Generated ER diagram for drawing part of Xfig.

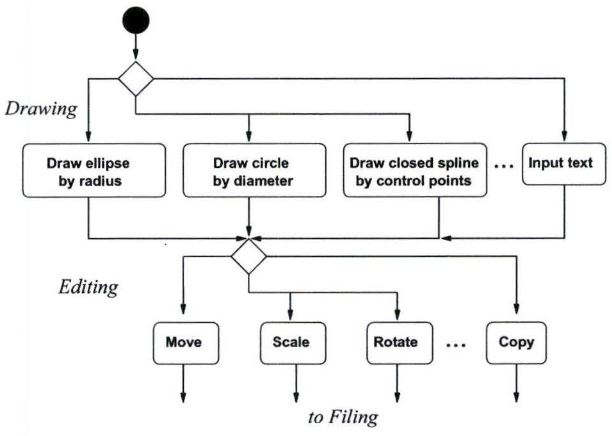


Figure 7.3: Generated activity diagram for drawing and editing parts of Xfig.

7.1.3 Design diagram generation

After populating the objectbase in scenario decomposition step, the information in the object base is transformed into design diagrams using the guidelines discussed in Section 5.5. Figure 7.2 illustrates parts of the generated ER diagram for "drawing" part, and Figure 7.3 illustrates the activity diagram for "drawing" and "editing" parts of Xfig tool.

7.2 Combined static-dynamic architecture recovery

In this section, we present the results of applying the proposed combined static and dynamic analyses architecture recovery on Xfig tool. To illustrate the difference between the results of static-only and combined static-dynamic recovery techniques, we compare the results of both techniques on an identical pattern query. In the following, applying the three steps of static pre-processing, dynamic pre-processing, and iterative pattern matching on Xfig is presented.

7.2.1 Static pre-processing

In this step, we use Refine C parser to parse the source code of Xfig and translate it to Abstract Syntax Tree (AST). Then we use Apriori data mining algorithm to transform the AST to source graph. Using the same data mining algorithm associated groups are found which leads to finding a similarity value between every two entities in the system. Pair-wise similarities between all the entities of the system are presented as a single similarity matrix. As mentioned in Section 6.3, for each entity in the system a search domain is 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. Because of memory and time limitations, in some cases we have to cut very large domains, to reduce the complexity of the search algorithm in the pattern matching step. In order to keep the relatively more similar entities in one's domain, we sort the domains of the entities decreasingly based on the similarity value before any cutting be done. Table 7.1 illustrates a part of the domain and the corresponding similarities of "init_draw" function in Xfig system.

Table 7.1: Part of the sorted domain and similarity values for "init_draw" function.

entity	**domain/similarity**					
init_draw	init_rotate	init_flip	redisplay_objects	redisplay_arcobjects	redisplay_line	...
	6.0	6.0	3.5	3.0	2.0	...

7.2.2 Dynamic pre-processing

For the dynamic analysis step, we use GNU gprof profiler to instrument the software code and capture the execution profiles. In this experiment we specify four features of the Xfig tool, namely "draw rectangle", "move", "rotate", and "copy", as the set of our frequent features. In order to eliminate the probable effects of the order in the execution of the features, instead of generating a single scenario, we generate a set of scenarios that contain different possible permutation of the order of the features to be covered. Trivially, the "draw rectangle" feature should always be executed first; but the three other features can be executed interchangeably. Therefore, with guidance of the set of scenarios generated for "drawing" and "editing"[1] parts of Xfig and the generated activity diagram (Figure 7.3) in the design view recovery stage, we define the set of scenarios presented in Figure 7.4 to cover all the permutations of the aforementioned features. We execute each of these scenarios and extract the frequencies for function-calls from their executions profiles. As a result, the final frequency assigned to each function-call would be the highest frequency of its appearance within all the profiles.

Table 7.2 illustrates a part of the execution profile of the first scenario in Figure 7.4 for two functions of Xfig system. The final frequencies of function-calls are then embedded into the source graph as the $freq$ attribute of the corresponding edges.

[1]The scenarios specific for "editing" part of Xfig are not shown in this case study.

#	**Scenario**
1	*"Draw rectangle, move it, rotate it, and flip it."*
2	*"Draw rectangle, move it, flip it, and rotate it."*
3	*"Draw rectangle, rotate it, move it, and resize it."*
4	*"Draw rectangle, rotate it, resize it, and move it."*
5	*"Draw rectangle, resize it, move it, and rotate it."*
6	*"Draw rectangle, resize it, rotate it, and move it."*

Figure 7.4: Generated scenarios for a particular set of features of Xfig.

Table 7.2: Part of the execution profile of the first scenario in Figure 7.4.

caller	**called/frequency**					
init_flip	flip_line	init_flipline	redisplay_line	flip_search	do_object_search	...
	5	2	1	1	1	...
setup_panel	set_rulermark	generate_pixmap	check_action	init_fill_pm	init_fill_gc	...
	992	17	2	1	1	...

7.2.3 Iterative pattern matching

In the pattern matching stage the Alborz tool provides a list of main-seed suggestions. Typically, these main-seeds are the entities that generated a higher average similarity with entities of their domain. However, main-seed selection can be totally random. From the suggested main-seeds we selected two of them, namely "init_rotate" and "init_update", that had also appeared in the execution profiles in the behavior view recovery. As mentioned in the beginning of this chapter, to be able to compare the result of our proposed recovery technique, we recover the components once with use of static information only, and once with combined static-dynamic information. Figure 7.5 illustrates the result of the architecture recovery of two components based on the two mentioned main-seeds and with size of 7 (left) and 11 (right). Figure 7.5.a shows the result of recovering the components with static information only. The main-seeds are shown as shaded nodes. As you see there are three links between the two components that do not have dynamic frequencies assigned to them in this case. Average similarities of each component which show the cohesiveness of the components are shown in the figure. Figure 7.5.b shows the same result as in Figure 7.5.a, but it also shows the dynamic frequencies of the edges connecting these two components, and the relative dynamic frequency of interaction of the two components, which implies the degree of dynamic coupling of the two components. Figure 7.5.c shows the result of performing component recovery with the same pattern query used for Figure 7.5.a, but by using both static and dynamic information from system. As it is shown in this figure, a number of entities of each component have changed compared to Figure 7.5.a, but the most important difference is movement of "draw_line" function from the right component to the left component. This change is because of the relatively high dynamic frequency of the edge connection functions "rotate_search" and "draw_line". The new similarity values and relative dynamic frequencies are once again shown in Figure 7.5.c. As you can see, the cohesion of each component has decreased by a factor of less than 10%. On the other hand, the dynamic coupling of the two components has decreased by a factor of greater than 170%. Therefore, the replacement of entities between components can be treated as a trade-off between higher intra-component cohesion and lower inter-component dynamic coupling, which can be decided of based on the objectives of a specific component recovery practice.

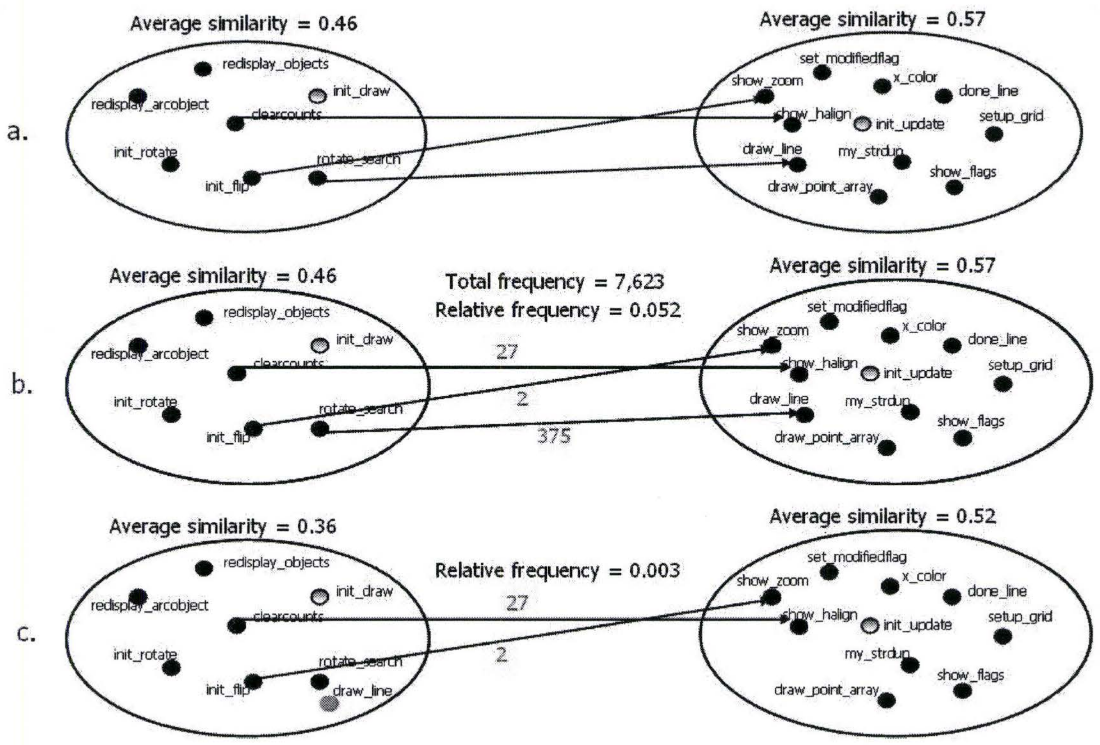


Figure 7.5: a. Recovered components based on static information only. b. Recovered components based on static information only, and dynamic information being demonstrated. c. Recovered components based on both static and dynamic information.

Chapter 8

Discussion and conclusion

In this thesis, we presented a novel multi-view framework to recover three views of a software system (i.e., design, behavior, and structure) where task scenarios were core entities to connect the tree views together. The design view generation is based on a systematic approach to define a set of task scenarios that conform with a scenario structure. The scenarios are then mapped onto a scenario domain model and the resulting instances are transformed into design diagrams using a number of guidelines. The behavior view is built on the analysis of the execution profiles that are the result of executing a set of scenarios that cover a set of frequent features of the system. The result of this analysis is a mapping between the function-calls of the system and an integer number that is the frequency of their call in the execution of the set of scenarios. The resulting frequencies are then embedded into the source graph representation of the system as an additional attribute that reflects the result of dynamic analysis in the architecture recovery process. Finally, in the structure view recovery, a set of conceptual components with corresponding constraints are defined for the tool as an architectural pattern query using XML notation. The recovery of these components is modelled as an instance of Valued Constraint Satisfaction Problem

(VCSP). The solution to the problem would be a set of concrete components that satisfy the specified constraints and have a minimal cost. The multi-view environment has been built in a toolkit called Alborz as a plug-in application for the Eclipse software development environment.

The proposed multi-view recovery has challenging issues to be dealt with. In the design view recovery, dealing with the ambiguity of the natural language makes the scenario decomposition a highly user-oriented task. In the behavior recovery, the software instrumentation tools usually produce very large execution profiles that need to be pruned from noise and recursion-based frequencies. Finally, in structure view recovery, defining pattern queries that lead to a semantically meaningful set of concrete components requires a good knowledge of the software system domain.

Bibliography

[1] Introductory tutorial on w3c xml schema from w3schools. http://www.w3schools.com/schema/default.asp.

[2] Rigi. http://www.rigi.csc.uvic.ca/rigi/rigiindex.html.

[3] Xfig version 3.2.3. http://www.xfig.org/.

[4] Sei software architecture definitions. http://www.sei.cmu.edu/architecture/definitions.html.

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

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

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

[8] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[10] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[11] Lawrence Chung and Kendra Cooper. A knowledge-based cots-aware requirements engineering approach. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 175–182, New York, NY, USA, 2002. ACM Press.

[12] C.Potts. Scenic: A strategy for inquiry-driven requirements determination. In *Proc. RE'99: International Symposium on Requirements Engineering*, Limerick, Ireland, June, 1999.

[13] Jules Desharnais, Ridha Khedri, and Ali Mili. Representation, validation and integration of scenarios using tabular expressions. *Journal of Formal Methods in Software Development. Special issue on tabular expressions*, 2002.

[14] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Derivation of feature component maps by means of concept analysis. Fifth European Conference on Software Maintenance and Reengineering, March 2001.

[15] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29:210 – 224, March 2003.

[16] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

[17] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *SEKE '02: Proceedings of*

the 14th international conference on Software engineering and knowledge engineering, pages 447–454, New York, NY, USA, 2002. ACM Press.

[18] 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.

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

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

[21] 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.

[22] Haumer, P.K. Pohl, and K. Weidenhaupt. Requirements elicitation and validation with real world scenes. In *IEEE Transactions on Software Engineering 24*, pages 1036–1054, 1998.

[23] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, November 1996.

[24] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. Saam: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994.

[25] Rick Kazman and Marcus Burth. Assessing architectural complexity. In *Proceedings of the CSMR*, pages 104–112, 1998.

[26] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstruction software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2):107–138., April 1999.

[27] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[28] Bruno T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–503, May 1998.

[29] 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, 1995.

[30] E. Nasr, L. McDermid, and G. Bernat. Eliciting and specifying requirements with use cases for embedded systems. In *In Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS?2)*, pages 350–357, January 2002.

[31] B. A. Nuseibeh and S. M. Easterbrook. Requirements engineering: A roadmap. In *In A. C. W. Finkelstein (ed) "The Future of Software Engineering ". (Companion volume to the proceedings of the 22nd International Conference on Software Engineering, ICSE'00). IEEE Computer Society Press*, 2000.

[32] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-*

Oriented Technologies and Systems (COOTS '98), pages 219–234. USENIX, 1998.

[33] Jolita Ralyte. Reusing scenario based approaches in requirement engineering methods: Crews method base. In *REP'99)*, pages 305–309, 1999.

[34] Reasoning Inc., 700 E. El Camino Real, Mountain View, CA 94040, USA. *Software Development Kit: Refine/C User's Guide for Version 1.2*, April 1998.

[35] Tamar Richner and stephane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 13, Washington, DC, USA, 1999. IEEE Computer Society.

[36] Claudio Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the IEEE CSMR'02*, pages 47–55, 2002.

[37] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *In Proceedings of the IEEE International Workshop on Program Comprehension (IWPC01)*, pages 115–116, Toronto, Canada, May 2001.

[38] 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.

[39] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching.* PhD thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2003.

[40] Kamran Sartipi and Kostas Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 408–419, Florence, Italy, November 2001.

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

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

[43] Kamran Sartipi, Kostas Kontogiannis, and Farhad Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of the IEEE IWPC*, pages 37–47, Limerick, Ireland, June 2000.

[44] Thomas Schiex, Helene Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the IJCAI-95*, pages 631–637, 1995.

[45] Michael Siff and Thomas W. Reps. Identifying modules via concept analysis. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 170–179, Washington, DC, USA, 1997. IEEE Computer Society.

[46] A. Sutcliffe. Scenario-based requirements engineering. In *Proceedings, 11th IEEE International Requirements Engineering Conference (RE'03)*, pages 320– 329, Monterey Bay, USA, 8-12th September 2003. IEEE Computer Society Press.

[47] A. G. Sutcliffe. Scenario-based requirements analysis. *Requirements Engineering Journal*, 3(1), 1998.

[48] Yiausyu Earl Tsai, Hewijin Christine Jiau, and Kuo-Feng Ssu. Scenario architecture - a methodology to build a global view of oo software system. In *COMPSAC*, pages 446–451, 2003.

[49] Vassilios Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

[50] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 122–132, 2004.

[51] Aline Vasconcelos, Rafael Cepeda, and Cla'udia Werner. An approach to program comprehension through reverse engineering of complementary software views. In *PCODA 2005: Program Comprehension through Dynamic Analysis*, pages 58–62, Pittsburgh, Pennsylvania, USA, 2005. IEEE Computer Society.

[52] W. Wang, S. Hufnagel, P. Hsia, and S. M. Yang. Scenario driven requirements analysis method. In *Proceedings of the Second International Conference on Systems Integration*, pages 446–451, Morristown, NJ, June 15-18 1992.

[53] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[54] Steven G. Woods and Qiang Yang. Program understanding as constraint satisfaction. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 314–323, 1995.

[55] Jingwei Wu and Margaret-Anne D. Storey. A multi-perspective software visualization environment. In *Proceedings of the CASCON conference*, pages 41–50, 2000.

[56] J. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, page 26(3):276?92, 1987.

[57] John. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[58] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 134–142, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Hong Hui Zhang and Atsushi Ohnishi. A transformation method of scenarios from different viewpoints. In *APSEC 2004*, pages 492–501, 2004.

[60] H. Zhu and L. Jin. Automating scenario driven structured requirements engineering. In *Proc. of COMPSAC'2000*, pages 311–318, Taipei, Taiwan, Oct. 2000. IEEE Computer Society Press.