An Orchestrated Multi-view Software Architecture Reconstruction Environment*

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

Abstract

Most approaches in reverse engineering literature generate a single view of a software system that restricts the scope of the reconstruction process. We propose an orchestrated set of techniques and a multi-view toolkit to reconstruct three views of a software system such as design, behavior, and structure. Scenarios are central in generating design and behavior views. The design view is reconstructed by transforming a number of scenarios into design diagrams using a novel scenario schema and generating an objectbase of actors and actions and their dependencies. The behavior view is represented by different sets of functions that implement different features of the software system corresponding to a set of feature-specific scenarios that are derived from the design view. Finally, the structure view is reconstructed using modules and interconnections that are resulted by growing the core functions related to the software features that are extracted during the behavior recovery. This orchestrated view reconstruction technique provides a more accurate and comprehensive means for reverse engineering of a software system than a single view reconstruction approach. As case studies we applied the proposed multi-view approach on two systems, Xfig drawing tool and Pine email system.

KEYWORDS: Recovery; Multi-view; Design; Behavior; Structure; Scenario; Schema; Pattern mining.

1. Introduction

Software systems and their platforms become costly to maintain for several reasons, such as: aging; lack of updated documentation; error-prone operation caused by patches and feature improvements; cease of platform support from the provider; and adoption of new technologies. No matter what the cause is, the target system would turn into a legacy system where in most cases the organizations are doomed to perform a maintenance operation (reverse- / re-engineering) to keep their software system operational. In this context, precise understanding of the design, behavior, and structure of the system, as different views of the software systems, are crucial in maintaining the software assets.

Software architecture views are the result of applying separation of concerns on a software engineering activity such as software development or reverse engineering. Reverse engineering research community have recently paid more attention to amalgamation of static and dynamic aspects of software systems [16, 10, 9] that extract structure and behavior views, respectively. Nevertheless, static analysis at different levels of abstraction (e.g., source code analysis, or architectural recovery) is still considered as the main focus for maintenance activities. One main reason for this is the completeness of the static information about the system 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 [5]. This valuable information usually can not be obtained easily by static analysis. In addition to the aforementioned two views, 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. Design view, when integrated with structure and behavior views provide a guided architectural reconstruction process.

In this paper, we propose a multi-view framework to recover three views of a software system that are orchestrated by the task scenarios. The multi-view model establishes the relations between the three views: design, behavior, and structure, where task scenarios are the key elements for collaboration of the three views. In the presented framework, a set of scenarios are generated using the evidences such

*This work was funded by Natural Sciences and Engineering Research Council (NSERC) of Canada.

as system documentation or user-system interaction. The scenarios are parsed to generate a design view of the software system. For the behavior view recovery, the user investigates the design view and selects specific features and corresponding scenarios to identify the groups of core functions that implement these features. The application of a sequential pattern discovery technique generates execution patterns as the means to identify the implementation of the software features within the source code. In a further step, each group of core functions that implement a feature will be used in the structure view recovery in order to generate one or more cohesive modules of functions using an optimization clustering algorithm and data mining association properties.

The proposed multi-view approach in this paper will assist the maintenance activities of software systems in different ways, such as: i) providing deep insight into the design properties of the implemented software system; ii) assessing structural merit of a software system through feature scattering among system files [13]; and iii) defining precise links among the software features in design view and their implementations in system files. The contributions of this paper include the reverse engineering of three orthogonal views of a software system, i.e., design, behavior, and structure, by the means of task scenarios to orchestrate and relate the results of a view to extract the relevant information from another view. The proposed multi-view framework is supported by a toolkit namely *Alborz* [14] that is built within the Eclipse plug-in environment.

The organization of the paper is as follows. Section 2 provides the related literature with respect to multi-view recovery. In Section 3 the proposed multi-view model and process are discussed. In Sections 4 and 5 and 6 the employed techniques for recovery of the three views: design, behavior, and structure, are presented. Section 7 presents a case study on the Xfig drawing tool. Finally, Section 8 concludes the paper.

2. Related work

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

Vasconcelos et al. [17] present a dynamic analysis 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. These views complement the extracted static view through integration within the Odyssey environment [2].

Riva et al. [9] 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 in architectural level. Their technique allows for creating domain-related architectural views for the architecture description of the system. Similarly in our approach, we use scenarios with design-derived features to guide the multi-view recovery process.

In a similar context, Deursen et al. [16] propose a view-based software reconstruction framework that provides a common framework for reporting reconstruction experiences and comparing reconstruction approaches.

Richner et al. [8] propose an approach to extract static and dynamic views from Java programs. The static view is generated from class files and visualized using Rigi reverse engineering environment. 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.

Overall, the significance of our technique is evinced in that we generate three views of a software system (design, behavior, and structure) where the task scenarios play a key role by guiding the multi-view recovery process.

3. Proposed multi-view framework

The proposed framework consists of a *multi-view model* and a *multi-view process* that are illustrated in Figure 1 and Figure 2, respectively. The multi-view model represents the relations between the three views *design*, *behavior*, and *structure* in a class diagram, where *scenarios* are the key elements for extraction and collaboration of the three views. The multi-view process in Figure 2 illustrates the overall mechanism to extract three views of a software system. In this 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 structure of scenarios must conform with a regular expression syntax. The structured scenarios are parsed to generate a design view of the software system that is represented by two types of diagrams, *entity-relationship diagram (E-R)* and *activity diagram*. These diagrams represent the implemented functionality and the major system data that are manipulated by the activities. For the behavior view recovery, the user investigates the design view and selects specific features in order to be used by the behavior recovery process. For each specific feature, a set of task scenarios are defined that share that feature. The execution of this set of scenarios on the *instrumented*[1] software system generates execution traces that are processed to extract *execution patterns*. Each execution pattern is a sequence of source code function calls that are

[1] Instrumentation refers to the process of inserting particular pieces of code into the software system (source code or binary image) to generate a trace of the software execution.

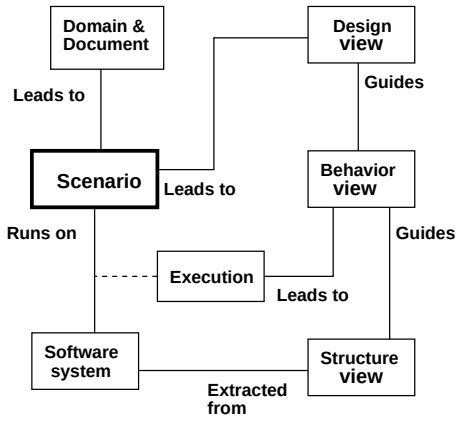

Figure 1. Proposed *multi-view model* **(class diagram) representing the relations between three views.**

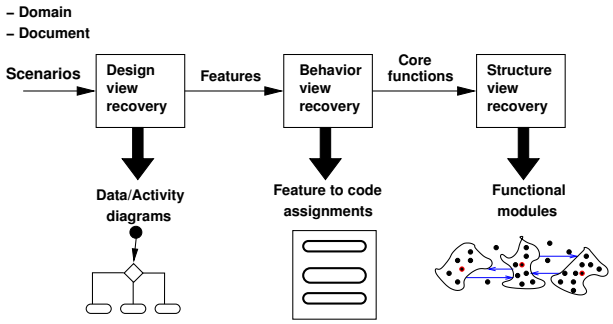

Figure 2. Proposed *multi-view process* **to extract three views of a software system.**

common among the execution of all scenarios in this scenario set. Iterating this process for a collection of features allows us to identify the realization of software features in the source code, namely *core functions*. Finally, each group of core functions that implement a feature will be used as a core of a cluster in the structure view recovery in order to produce a larger group of cohesive functions. The structure view takes advantage of the association relations among the functions to determine the proximity of other functions with the core of each cluster that consequently generates clusters that represent the software components.

The overall process in this multi-view framework allows us to relate abstract design diagrams to the concrete implementation of the functional elements of the design view. It should be noted that each view in this work (design, behavior, structure) has its own application that will be discussed in the corresponding section. However, this paper focuses on the collaboration among these three views that is orchestrated by the task scenarios.

4. Design view generation

In this section, we discuss the steps for transforming the knowledge embodied in the text of task scenarios into design related information represented by two types of diagrams, i.e., entity relationship diagram (E-R) and activity diagram, using the process illustrated in Figure 4. In a nutshell, the design generation approach generates and structures a set of task scenarios and then uses a novel *scenario domain model* to parse the composed scenarios into ingredients of the adopted design diagrams. The proposed process consists of three steps, as follows.

Design step 1 *(scenario generation)*. Task scenarios are the main elements that orchestrate the proposed multi-view architecture recovery framework. In this paper, we adopt a structured text-based representation for scenarios that conform with the regular expression syntax in Figure 3. In this scenario syntax each scenario consists of a sequence of one or more *Actors*, *Actions*, and *Working Information*, each of which can have between zero or more *Constraints* that will be defined in Design step 2 below. In this form we can generate syntactically correct scenarios which will be further decomposed using the scenario domain model to populate the objectbase in Design step 2 and generate design diagrams in Design step 3. These scenarios will be further used to populate a knowledge-base of *scenario templates* to reuse the captured business rules in a future similar case. The sources for scenario generation are available evidences such as: system user interface, user-manual, and expert-user's knowledge of the system.

Design step 2 *(scenario decomposition)*. The class diagram representation of the proposed scenario domain model is presented in Figure 5. This domain model is intended to provide the classes of information in scenarios from different application areas. We have applied this model on three systems, including a fast-food restaurant system, Xfig drawing tool, and an automatic banking machine (ABM) system. The text of the structured scenarios is parsed using this domain model and the resultant instances of classes in the domain model are stored as a record in the objectbase. The schema of the objectbase has an entry for each class of the domain model as well as an index entry as its primary key.

As shown in Figure 5, in this 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* and *Constraint* classes.

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

Figure 3. Regular expression syntax for scenario generation, where "+" and "0..M" ("1..N") represent composition and range, respectively.

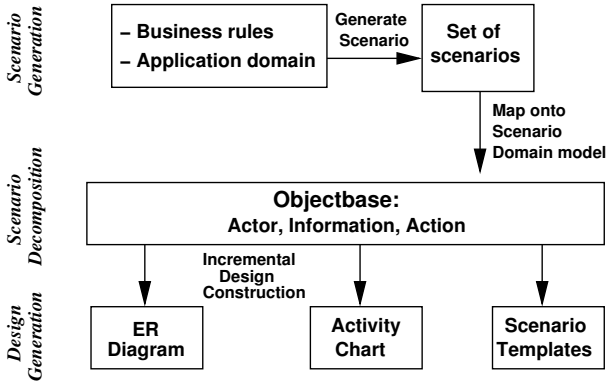

Figure 4. Design view generation based on task scenarios and scenario domain model.

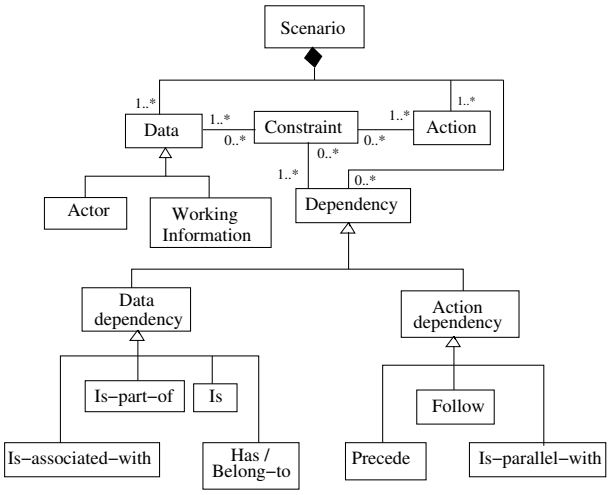

Figure 5. Scenario domain model that is used to parse a scenario and populate an objectbase.

Below, the classes of the proposed scenario domain model are introduced:

- *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.
- *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 is explained shortly. Actions can be categorized into three different types of *Input, Internal,* and *Output*, based on the scope of their working information manipulation.
- *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.
- *Dependency*: a dependency refers to a relation between two instances of the classes *Actor, Action*, and *Working information*. During parsing 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. A dependency can be of type *Data dependency* or *Action dependency*.

The proposed scenario domain model in Figure 5 includes a *Constraint* class. This class contains information about the 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.

Design step 3 (*design generation*). In this step, Entity-Relationship and Activity diagrams are generated using the instances of the classes of the domain model that are stored in the objectbase.

- *Entity-Relationship diagram*. Instances of *Actor* and *Working information* classes in the objectbase are candidates for entities and their corresponding attributes in E-R diagram; and instances of different subclasses of *Data dependency* are candidate relationships that connect different entities and assign attributes to entities.
- *Activity diagram.* Instances of *Action* class are candidate activities in the activity diagram; and the in-

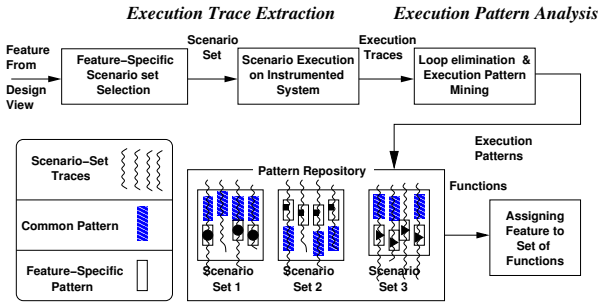

Figure 6. Behavior view recovery based on execution patterns to identify feature functionality in the source code.

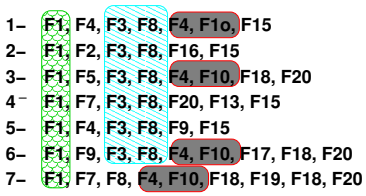

Figure 7. A collection of 7 execution traces. Different types of shaded areas correspond to three execution patterns.

stances of different subclasses of *Action dependency* relate other elements of the activity diagram such as: edges, diamonds, joins, and forks.

Due to space limitation the details of generating E-R and activity diagrams are not presented in this paper.

The proposed design view generation from task scenarios is a systematic approach to transform the informal information from task scenarios into well-formed design diagrams. In the next section, we discuss how the diagrams of design view generation are used to provide features and scenario sets to be fed into behavior recovery of the system.

5. Behavior view recovery

Figure 6 illustrates the steps for recovery of a software system's behavior view as a means to identify the realization of the software features by the source code functions. The process for behavior view recovery exploits the relation discovery power of data mining techniques. The steps of this process are as follows.

Behavior step 1 (*execution trace extraction*). As mentioned in Section 3, important features of a software system are identified as a result of the design generation process. The generated activity diagram assists us to define a set of relevant scenarios that examine a single software feature. We call this set of scenarios a *feature-specific scenario set*. For example, in the case of a drawing tool software system, a group of scenarios that share the "*move*" operation to move a drawn figure on the computer screen would constitute such a feature-specific scenario set. Next, the software system is instrumented in order to produce execution traces when a task scenario is executed on the system. A major obstacle in run time analysis of a system is the large size of the generated execution traces that makes the task of analysis a daunting one. The effective trace of functions for the intended scenario is cluttered by a large number of function calls from operating system, initialization / termination operations, utilities, and also repetitions that are caused by the program loops. To make the large size of the generated traces manageable, we remove all redundant function calls caused by the program loops. This operation contributes in scalability of the approach.

In the remaining of this section, we describe the application of sequential pattern mining technique to discover groups of functions in execution traces that correspond to certain system features. In the data mining literature, the sequential pattern mining technique is used to extract frequently occurring patterns among the sequences of customer transactions [3]. In this paper we adopt the above technique in order to extract frequently occurring patterns of function calls among program execution traces. The extracted sequential pattern consists of *only* contiguous parts of an execution trace and the interleaving of other functions within the patterns are not allowed. This characteristic produces meaningful *execution patterns* that correspond to core functions implementing specific features of the system.

In the proposed approach we distinguish between two types of patterns: *common execution patterns* and *feature-specific execution patterns*. The categories of the execution patterns and the proposed extraction mechanism in our approach are briefly discussed below:

- *Common execution pattern*. A common pattern exists in the majority of feature-specific scenario sets that are executed on the system. In order to extract such a pattern, we should use a filtering mechanism to filter out the feature-specific patterns from this group of patterns. An example of a common pattern is the pattern of function calls that is produced by initialization component of each program execution.

- *Feature-specific execution pattern*. Each pattern in this category corresponds to the core functions that implement the targeted feature of a feature-specific scenario set. Such a pattern exists in the majority of traces in a feature-specific scenario-set. As mentioned above, the

common patterns are also extracted along with feature-specific patterns. The separation of these two types of patterns are discussed later in this section.

Figure 7 illustrates a collection of seven execution traces and their corresponding extracted execution patterns that include both feature-specific and common patterns.

Behavior step 2 (*execution pattern analysis*). We employ a strategy to focus on the execution patterns corresponding to specific features within the scenario sets. In this context, we exploit the structural characteristics of the concept lattice [15] in order to separate the functions exclusive to a specific feature from the group of functions that implement the common features. In the context of a concept lattice, attributes that are extensively shared among most of the objects appear in the upper region of the lattice and vice versa. In our setting for concept lattice analysis, an object is a feature of a feature-specific scenario set and an attribute is a function that participates in the execution patterns of this feature-specific scenario set. Since common functions are executed through almost every feature-specific scenario set, these functions appear in upper region of the lattice. On the other hand, functions that are exclusive to certain features of the software are located in lower region of the lattice. As a result, the core functions that exclusively implement certain features of the system are identified.

In addition to the main application of the proposed behavior recovery, i.e., identifying the software feature implementation in the source code, this approach has been used for: i) measuring the scattering of the software features among the structural modules; ii) assessing the structural cohesion of the software modules; and iii) visualizing the functional distribution of specific features on a lattice [13]. In the next section, we discuss how the result of the behavior view is used to provide semantics to the structure view recovery of the system.

6. Structure view recovery

Figure 8 illustrates the steps for recovery of a software system's structure view that generates cohesive software modules from source code functions. The process for structure recovery consists of two major steps as *fact extraction* and *module reconstruction* that are briefly described below. We use Alborz reverse engineering toolkit [14] that provides a supervised optimization clustering technique.

Structure step 1 (*fact extraction*). In the fact extraction step, the software system is parsed to generate an *abstract syntax tree* (AST) that contains all the constructs corresponding to the programming language of the software

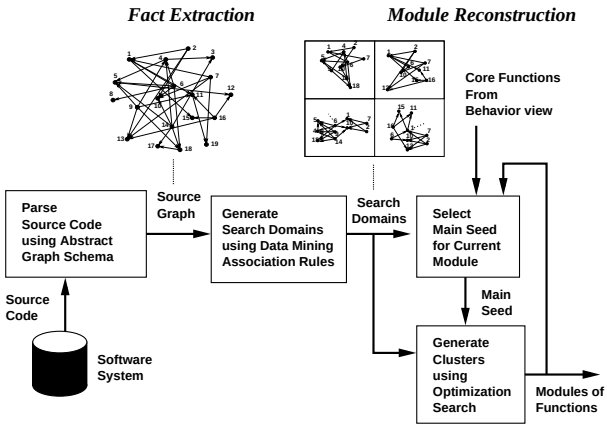

Figure 8. Structure view recovery based on maximal association and optimization clustering.

system, e.g., C. Using a *graph schema* [11] that defines architectural level entities for recovery of software modules (i.e., function, aggregate types, and global variables)[2] and their relationships (i.e., function-call, type-use, and variable-use), we produce a graph representation of the software system (namely *source graph*) and store it in a fact base. Considering the number of entities in a medium size software system (usually more than 1000 entities), searching the whole search space (source graph) is an intractable problem. Hence, we must restrict the search domain for each module to a group of eligible entities. We apply association-based data mining algorithms on the source graph and consequently decompose the source graph into smaller regions (search domains) where each search domain consists of a number of entities that are associated with an entity in that domain, namely *main-seed*. We then restrict the search for each module to one (or more) of these regions.

Structure step 2 (*module reconstruction*). We perform a supervised optimization clustering technique that incrementally generates software clusters as cohesive modules of functions that are interconnected through function imports and exports. Each module consists of one or more main-seeds as the core functions of the module and a sub-optimal version of the A^* search algorithm is used to collect the group of highly associated functions into a module. The search space for a module is restricted to the functions in the search domain(s) of the corresponding main-seed(s). We define an association-based similarity metric that is based

[2]The focus of this work is on the function-level analysis, however the toolkit that we use can perform file-level analysis as well.

on the group of entities with maximum association property. The maximal association property is defined in the form of a maximal set of entities that all share the same relations to every member of another maximal set of entities. Hence, these groups possess a higher-degree of cohesiveness and are suitable to form clusters based on an entity association similarity metric. Such a similarity metric encodes the structural property of the groups of maximally associated entities.

The structure recovery without the collaboration of the proposed multi-views relies on the tool-provided facilitates that generate an ordered list of highest qualified main-seeds for the next module recovery. While this method produces very high cohesive modules of functions, it can not produce meaningful modules since the core functions are selected based on the static structural properties and not the functionality of the main-seeds. Whereas, the proposed multi-view collaborative approach provides core functions as the module main-seed(s) that implement meaningful software features. Furthermore, these software features have been derived from design diagrams that stem from the software's functional requirements. In Figure 8, the core functions from behavior view are used to produce semantically meaningful clusters as system components.

Clustering techniques can be classified as "automatic or supervised" techniques [7] and "user-centric" techniques [12]. These techniques attempt to restructure the original system into a new system whereby it is composed of higher quality modules in terms of high-cohesion and low-coupling characteristics. A clustering technique must meet specific requirements of the new environments for software architecture recovery, including: user involvement to guide the process according to domain/document knowledge [6]; incremental recovery to allow partial system recovery [7]; and hierarchical decomposition to deal with complexity and tractability issues inherit to the analysis of large systems [6]. The clustering technique proposed in this paper acts as an effective user assistant by providing an adequate level of automation and useful information so that the user can investigate and control the recovery process.

7. Case study

In this section, we describe the results of applying the proposed multi-view framework on two medium-size open source software systems, Xfig drawing tool and Pine email client system[3]. Because of space limitation in this paper we discuss Pine system for a statistical comparison with Xfig; however, the full analysis is presented only for Xfig. We use the Alborz reverse engineering toolkit [14] in our experiments to study the architecture of Xfig 3.2.3d [1] which is an open source, medium-size (80 KLOC), menu driven, C language drawing tool under X Window system. Xfig is used to draw and manipulate graphical objects (circle, ellipse, line, spline, rectangle, and polygon) through operations such as copy, move, delete, edit, scale, and rotate. In the following, each step of our experiments is described with respect to the framework in Section 3 and the three views in Sections 4 to 6.

[3]We experimented with Pine 4.4.0 which is a medium-size (207 KLOC), open source system that is written in C.

#	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 9. Generated scenarios for drawing part of Xfig.

7.1. Design view generation

In this subsection we demonstrate the results of applying the three steps of design view generation described in Section 4.

Scenario generation. Figure 9 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 structure imposed by the scenario syntax in Section 4. For example, Scenario #1 in Figure 9 conforms with the scenario syntax, 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).

Scenario decomposition. Every generated scenario is parsed according to the scenario domain model in Figure 5 and the generated instances of the domain model classes are stored in the objectbase. As an example, the generated instances from decomposition of Scenario #1 are as follows:

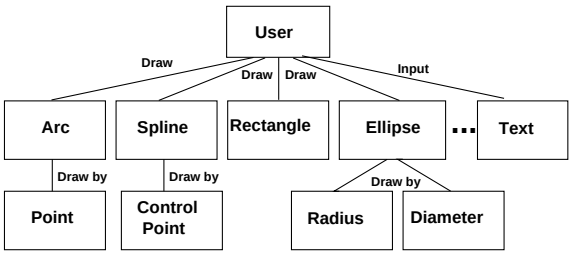

Figure 10. Generated ER diagram for drawing part of Xfig.

Decomposition of Scenario #1 in Figure 9

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

The same steps are repeated to cover the scenarios of the editing part of the Xfig. However, the details for this part are not presented here due to space limitation.

Design diagram generation. Figure 10 illustrate parts of the generated ER diagram for drawing, and Figure 11 illustrates the activity diagram for drawing and editing parts of Xfig tool. The detailed guidelines for generating ER and activity diagrams from the objectbase are not presented in this paper.

7.2. Behavior view recovery

In this subsection the results of applying the behavior recovery process on the Xfig drawing tool is demonstrated. In the activity diagram in Figure 11, each activity box corresponds to a certain feature of Xfig. In order to generate the feature-specific scenario set that targets a certain feature, we select the collection of all possible paths in the activity diagram that initiate from the start node and pass the specific feature. For example, the set of feature-specific scenarios for *draw ellipse by radius* feature includes scenarios such as: *"User draws ellipse by radius, and moves ellipse by radius."*, *"User draws ellipse by radius, and rotates ellipse by radius."*, and *"User draws ellipse by radius, and copies ellipse by radius."*

In a further step, we execute the resulting feature-specific scenario sets, and finally extract the execution patterns for each set. Table 1 presents the statistical information for both Xfig drawing system and Pine email client, where "*average trace size*" and "*average pruned trace size*" are in the range of tens of thousands, whereas the number of extracted patterns are in the range of tens (or hundreds). This signifies drastic reductions in the sizes of the traces to be analyzed after pattern mining operations.

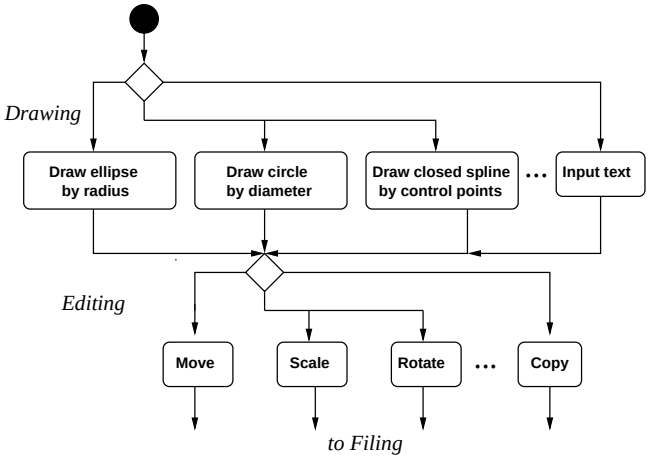

Figure 11. Generated activity diagram for drawing and editing parts of Xfig.

Figure 12 illustrates a part of concept lattice that represents six extracted Xfig features and their corresponding functions. The results of the feature to function mapping for the features in Table 1 are shown in Table 2.

7.3. Structure view recovery

The feature to function mapping presented in Table 2 is used to select main seeds for the clustering algorithm in this stage. In our case study, we selected each core function of the feature *drawing circle by radius* in the first row of Table 2 as a main seed. The clustering algorithm groups a set of cohesive functions around each seed.

Figure 13 illustrates the result of clustering a small group of Xfig functions into three modules (clusters) according to the feature *drawing circle by radius*. The extracted core functions for this feature presented in the first row of Table 2 represents the functions that are required to draw a circle. In this context, the drawing operation begins with selecting the specific shape (i.e., circle by radius) that is implemented by *circlebyradius_drawing_selected*. The operation continues by initializing a new circle, using *init_circlebyradius_drawing*, and terminates after creating a circle using the *create_circle_byrad* function.

In Figure 13 the main-seeds of the clusters have been highlighted. In this case, cluster 2 has one main-seed and cluster 1 and cluster 3 contain two main-seeds. This is because the main-seeds in cluster 1 (or 3) have generated highly overlapped clusters of functions as an evidence that their clusters must merge. A close investigation of the generated clusters indicates that each cluster consists of func-

System	Specific Feature of Xfig	Number of Different Scenarios	Average Trace Size	Average Pruned Trace Size	Number of Extracted Patterns	Average Pattern Size
Xfig	Circle-Diameter	10	7234	2600	46	33
	Circle-Radius	10	8143	2463	48	32
	Ellipse-Diameter	10	6405	2536	41	37
	Ellipse-Radius	10	7351	2549	39	35
Pine	Compose	3	58660	14930	46	529
	Folder List	4	22410	6743	24	491
	Message Index	5	39000	12760	43	345
	Address Book	3	59221	16024	70	212

Table 1. The result of execution trace generation and execution pattern mining for a collection of 4 features for Xfig drawing tool and for Pine email client system.

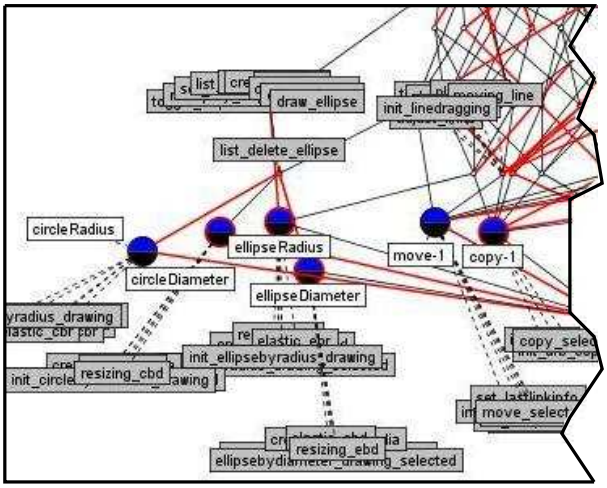

Figure 12. A Part of concept lattice representation of the specific features and their corresponding functions for the Xfig drawing tool.

Feature	Extracted Core Functions
Circle Radius	init_circlebyradius_drawing, elastic_cbr, resizing_cbr create_circlebyrad, circlebyradius_drawing_selected
Circle Diameter	init_circlebydiameter_drawing, elastic_cbd, resizing_cbd create_circlebydia, circlebydiameter_drawing_selected
Ellipse Diameter	init_ellipsebydiameter_drawing, elastic_ebd, resizing_ebd create_ellipsebydia, ellipsebydiameter_drawing_selected
Ellipse Radius	init_ellipsebyradius_drawing, elastic_ebr, resizing_ebr create_ellipsebyrad, ellipsebyradius_drawing_selected

Table 2. Extracted core functions corresponding to 4 specific Xfig features.

tions with similar naming convention to ensure that the modules are cohesive. The following interpretations can be made by investigating the three clusters. Cluster 1 collects functions that are required for initializing different shapes in Xfig; where all 16 functions in this cluster have been prefixed by the keyword *init* (as initialize). This is a strong indication of the cohesiveness of this module. Cluster 2 collects functions that are required for *selecting* different shapes. Cluster 2 has 10 functions where 9 functions have been post-fixed with *selected* that are used to select four types of figures *circle, spline, ellipse, and line*. Finally, Cluster 3 collects functions that implement drawing circle / ellipse by diameter / radius. The naming convention of the functions in these three clusters suggest that the design of Xfig is based on replication of functionality for similar features.

8. Conclusion

In this paper, we presented a novel multi-view framework to recover three views of a software system (i.e., design, behavior, and structure) that are orchestrated by the task scenarios as the key elements for extraction and collaboration of the three views. The design view generation is based on a systematic approach to define a set of task scenarios that are mapped onto a scenario domain model. The behavior view is built on the analysis of the execution traces that are extracted by running feature-specific task scenarios on the instrumented software system that generates a group of core functions that implement a specific feature. Finally, the group of core functions will be used in the structure view recovery in order to produce larger groups of cohesive functions that correspond to the targeted feature. The latter view provides a link between abstract elements and features in the design view to the implementation of those features in source code. The proposed approach incorporates pattern mining as an integral part of the structure and behavior view recovery. 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. The software instrumentation tools usually produce very large execution traces that need to be pruned from noise or loop-based patterns. Similarly, the discovery of the patterns using data mining algorithms usually generates an overwhelming

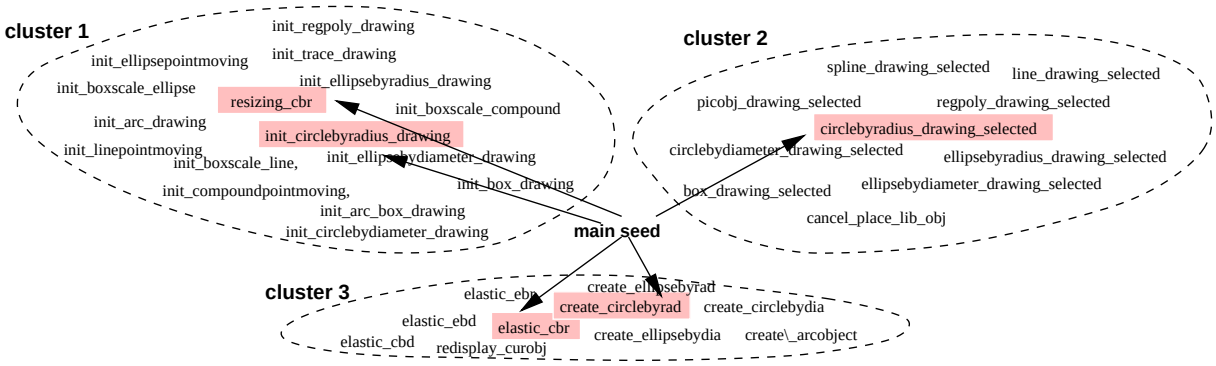

Figure 13. The result of clustering Xfig functions according to the feature *circle-by-radius***.**

number of patterns both in behavior and structural views, hence a post-processing may be required to obtain the major and sufficiently distinct patterns as the core functionality of the static/dynamic components.

References

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

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

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the IEEE International Conference on Data Engineering*, ICDE'95, pages 3–14, 1995.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29:210 – 224, 2003.

[5] M. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of ICSE Workshop on Dynamic Analysis*, WODA'03, 2003.

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

[7] R. Koschke. An incremental semi-automatic method for component recovery. In *Proceedings of the IEEE Working Conference on Reverse Engineering*, WCRE'99, pages 256–267, 1999.

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

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

[10] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proceedings of the IEEE International Conference on Program Comprehension*, ICPC'06, pages 84–88, 2006.

[11] K. Sartipi and K. Kontogiannis. On modeling software architecture recovery as graph matching. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM'03, pages 224–234, 2003.

[12] K. Sartipi and K. Kontogiannis. A user-assisted approach to component clustering. *Journal of Software Maintenance: Research and Practice (JSM)*, 15(4):265–295, July/August 2003.

[13] K. Sartipi and H. Safyallah. Application of execution pattern mining and concept lattice analysis on software structure evaluation. In *Proceedings of Software Engineering and Knowledge Engineering*, SEKE'06, pages 302–308, 2006.

[14] K. Sartipi, L. Ye, and H. Safyallah. Alborz: An interactive toolkit to extract static and dynamic views of a software system. In *Proceedings of the ICPC'06*, pages 256–259, 2006.

[15] M. Siff and T. W. Reps. Identifying modules via concept analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM'97, pages 170–179, 1997.

[16] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings of the IEEE Working Conference on Software Architecture*, WICSA'04, pages 122–132, 2004.

[17] A. Vasconcelos, R. Cepeda, and C. Werner. An approach to program comprehension through reverse engineering of complementary software views. In *Proceedings of IEEE Program Comprehension through Dynamic Analysis*, PCODA'05, pages 58–62, 2005.

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