SCAN: an Approach to Label and Relate Execution Trace Segments

Soumaya Medini[1], Giuliano Antoniol[1], Yann-Gaël Guéhéneuc[1], Massimiliano Di Penta[3], Paolo Tonella[2]

[1] DGIGL, École Polytechnique de Montréal, Canada
[2] FBK, Trento, Italy
[3] Dept. of Engineering, University of Sannio, Italy

***Abstract*—Identifying concepts in execution traces is a task often necessary to support program comprehension or maintenance activities. Several approaches—static, dynamic or hybrid—have been proposed to identify cohesive, meaningful sequence of methods in execution traces. However, none of the proposed approaches is able to label such segments and to identify relations between segments of the same trace.**

This paper present SCAN (Segment Concept AssigNer) an approach to assign labels to sequences of methods in execution traces, and to identify relations between such segments. SCAN uses information retrieval methods and formal concept analysis to produce sets of words helping the developer to understand the concept implemented by a segment. Specifically, formal concept analysis allows SCAN to discover commonalities between segments in different trace areas, as well as terms more specific to a given segment and high level relations between segments.

The paper describes SCAN along with a preliminary manual validation—upon execution traces collected from usage scenarios of JHotDraw and ArgoUML—of SCAN accuracy in assigning labels representative of concepts implemented by trace segments.

***Keywords*-Concept identification, dynamic analysis, information retrieval, formal concept analysis.**

I. Introduction

Many software engineering tasks require a preliminary activity which consists of identifying those program elements (e.g., classes, methods, etc.) that contribute to implement specific domain concepts, application features or computation phases [1], [2], as well as understanding the relations among them. A typical scenario where developers need to assign a concept to program elements is the following. Let us assume that (1) an unwanted behavior (*e.g.*, a failure) has been observed in a software system under certain execution conditions, and that (2) we can reproduce the failure by means of execution traces collected while executing the specific scenario leading to the failure. Such traces may be very large and manually analyzing them might be a very expensive, labor-intensive and error-prone task.

To alleviate the burden of manual analysis, different concept location approaches have been proposed in the literature, including static approaches [3], dynamic approaches [4], [5], and hybrid ones [6], [7], [8], [9], [10]. The hybrid approach proposed by Asadi *et al.* [9] and by Medini *et al.* [10] resorts to the textual content of the methods contained in execution traces to split traces into segments that likely participate in the implementation of the concepts related to the features of interest. The underlying assumption is that when a specific feature is executed within a complex scenario (*e.g.*, "Open a Web page from a browser" or "Save an image in a paint application"), the set of relevant methods being invoked is likely to be conceptually cohesive, decoupled from those of other features and invoked in sequence. The main drawback of such hybrid approaches [9], [10], as well as of dynamic approaches such as [4], is that the developer has little support in the phase of assigning concepts to trace chunks *i.e.*, attaching a label representative of the implemented concept.

This paper presents SCAN (Segment Concept AssigNer)—an approach inspired by previous work on aspect mining [5] and method summarization [11], [12], [13], [14]—to assign meaningful labels to chunks of segmented traces. SCAN labels trace segments by applying Information Retrieval (IR) techniques to terms extracted from method signatures. Then, to highlight relations (and differences) between segments, SCAN uses Formal Concept Analysis (FCA) on selected terms used to label each segment. Specifically, relations between segments and terms as well as between different segments can be discovered by inspecting the FCA lattice. Multi-threading is an important feature of modern processors, however, it induces variability in traces collected for the a given scenario. To limit the effect of multi-threading variability in assigning labels to segments, SCAN is able to merge segments obtained in multiple executions of the same scenario. Although SCAN is designed to complement previous work on trace segmentation [9], [10], SCAN is generic and can be applied to the output of any trace segmentation technique.

We have applied SCAN to assign labels to segmented traces produced by executing scenarios of two widely known Java applications, JHotDraw and ArgoUML. First, we have segmented traces with a tool set implementing the approach by Medini *et al.* [10]. Then, we have used SCAN to label each segment with a list of terms. To evaluate the quality

of the obtained labeling, we compared them with manually-assigned labels obtained by 1) inspecting the source code, 2) checking the available documentation, and 3) performing a step-by-step analysis of the execution traces. We performed a qualitative as well as a quantitative analysis. From a qualitative point of view in most cases SCAN suggested labeling terms are effective to help grasping the segment functionality. The quantitative analysis reveals a more variegated situation with encouraging results. We measured accuracy as precision and recall between manually assigned labels and automatic labels; recall median varies between 50% and 80%, while precision ranges between 30% and 70% depending from the application and scenario. In essence, at minimum 50% of term used to manually label the segment is retrieved by SCAN, supporting the claim that it can help program comprehension. Finally, qualitative analysis of the FCA lattices supports the claim that SCAN effectively detect segments relations.

II. Related Work

Commonalities can be found between this work and previous studies on concept identification or code (method) summarization. However, the latter is not the focus of our work, since our aim is not obtaining a description of a method valid for all its clients. We rather aim at extracting descriptive information that characterizes the contribution of a method to a specific execution scenario. To the best of our knowledge, no previous work considered the extraction of descriptive concepts from execution trace segments.

The work by Asadi *et al.* [9] is somehow more related to the current study as they present a concept location approach using a genetic algorithm. The previous work by Medini *et al.* [10] is an evolution toward scalability of the works by Asadi *et al.* [9]. In these works ([9] and [10]), authors identified concepts by finding cohesive and decoupled segments in a trace using a genetic algorithm and dynamic programming [10].

Pirzadeh *et al.* [15] proposed a trace sampling framework based on stratified sampling to reduce the size of a trace by distributing the desired characteristics of an execution trace similarly in both the sampled and the original trace. To generate sampled execution traces, random sampling techniques have been extensively used. Random sampling may generate in samples that are not representative of the original trace. This work was extended [16] by extracting higher-level views that characterize the relevant information about execution traces.

We share with the previous works summarized above the general idea of concept location and specifically of assigning a concept to a execution trace segment. However, this work extends such works and in particular the previous work by Medini *et al.* [10] with the aim of providing segment labeling, i.e., it performs the reverse engineering of segment-specific concepts from linguistic information associated with the execution scenario of interest.

Recently, Sridhara *et al.* [12] presented a novel technique to automatically generate comments for Java methods. They used the signature and the body of a method to generate a descriptive natural language summary of the method. The developer is left in charge to verify the accuracy of generated summaries. The work was extended and improved [13] by using a classification of code into fragments, to generate a natural language description of each fragment. The authors identified three types of fragments: sequence fragments, conditional fragments and loop fragments. We share with summarization work the goal of producing meaningful linguistic descriptions of program elements starting from the source code. However, our goal is different: we aim at obtaining a high-level description of a segment, possibly representative of a feature or concept implemented by the segment.

In general, we also share with many existing works the idea of using information retrieval as well as concept analysis. Information retrieval has been widely adopted in traceability recovery and feature location [17], [6]. Concept analysis proved useful in many software engineering tasks such as feature location [18], remodularization [19], aspect mining [5] and design patter identification [20], just to name a few. Here we use information retrieval and concept analysis as a tool to support linguistic information processing, with the aim of labeling trace segments.

III. Background Notions

In this section we briefly summarize key concepts and techniques used in this paper.

A. Trace Segmentation

This section summarizes the trace segmentation approach by Medini *et al.* [10]. As mentioned in the introduction, the approach aims at grouping together subsequent method invocations that form conceptually cohesive [21] groups.

Before segmenting the traces, the approach first prunes out—by analyzing invocation distributions—*utility* methods repeated in various trace regions *e.g.*, methods related to mouse events. Then, the trace is compressed using a run length encoding algorithm, to remove repetitions of method invocations.

On the filtered and compressed trace we model methods as documents, by collecting methods name, parameters and bodies. Each document (method) is processed as in information retrieval systems applied to software engineering [21]. We first extract terms from the source code, split compound identifiers separated by camel case (*e.g.*, `getBook` is split into `get` and `book`), remove programming language keywords and English stop words, and perform stemming [22]. We then index the obtained terms using the *tf-idf* indexing mechanisms [23]. We obtain a term–document matrix, and finally, we apply Latent Semantic Indexing (LSI) [24] to

reduce the term–document matrix into a concept–document[1] matrix, choosing, as in previous work [9], an LSI subspace size equal to 50. We used FacTrace to generate the concept–document matrix [25].

The final step consists in applying dynamic programming optimization techniques to segment the obtained trace. The cost function driving the search relies on conceptual (*i.e.*, textual) cohesion and coupling measures [21], [26], rather than structural cohesion and coupling measures. More precisely it attempts to balance segment cohesion, average (textual) similarity between the source code of any pair of methods invoked in a given segment, and coupling between a segment and all other segments in the trace. More details can be found in [9], [10].

B. Formal Concept Analysis

To identify relations between concepts identified in different segments, we use Formal Concept Analysis (FCA). FCA [27] groups *objects* that have common *attributes*. In this paper, objects are segments and attributes are terms extracted for the segments by means of IR techniques. The starting point for concept analysis is a *context*, i.e., a set of objects, a set of attributes, and a binary relation between objects and attributes, stating which attributes are possessed by each object. In our case, the binary relation states which term (attribute) is included by each segment (object). A *concept* is a maximal collection of objects that has common attributes, i.e., it is a grouping of all the objects that share a set of attributes, in our case, a cohesive set of segments sharing terms. More formally a concept is a pair of sets (X, Y) such that:

$$X = \{o \in O | \forall a \in Y : (o, a) \in P\} \quad (1)$$

$$Y = \{a \in A | \forall o \in X : (o, a) \in P\} \quad (2)$$

where O is the set of objects, A is the set of attributes and P is the binary *possess* relation between them. X is said the *extent* of the concept and Y is said the *intent*. To apply FCA, in this paper we used the Concept Explorer[2] tool.

IV. The SCAN Approach

SCAN consists of the following main building blocks: a segmentation merger, a component for relevant term identification, and a FCA module, to identify relations between segments. SCAN accepts as input one or more segmented traces. In this work, without loss of generality, we consider the trace segmentations obtained using the tool by Medini *et al.* [9], [10].

[1]In LSI "concepts" refer to orthonormal dimensions of the LSI space, while in the rest of the paper "concept" means some abstraction relevant to developers.

[2]http://conexp.sourceforge.net/

A. Segmentation Merger

The role of the segmentation merger is to recognize similarities between segments belonging to multiple execution traces and merge them. Because of multi-threading, of variations in application inputs (some of which not fully controllable by the software engineer instrumenting the application), and of variations in machine load condition, multiple executions of the same scenario may lead to different sets of segments, or even to related segments containing method invocations in different ordering (due to thread interleaving).

Let $S = (s_1, \ldots, s_n)$ and $Z = (z_1, \ldots, z_m)$ be two segmentations, (i.e., sequences of segments), the merger computes the $n \times m$ set of similarities between elements of the two sequences and associates pairs with a similarity higher than a given threshold. Let $n > m$, thus we have more segments in S, for each z_i the merger computes all similarities $\sigma(z_i, s_j)$ and keeps pairs above a given threshold. SCAN attempts to find both one to one as well as many to one relations between the shorter segmentation (likely containing larger segments) and the other one, containing more and (on average) shorter segments.

Highly-similar segments, of different traces for the same scenario, are supposed to contribute to the same functionality, regardless of the specific thread interleaving or trace area they occurred. The similarity between two segments is computed as the Jaccard coefficient between terms extracted from the method signatures. Specifically, the signatures of the methods contained in a segment are processed; terms extracted; and language types removed. The result is a set of terms and the higher the number of terms in common the higher will be the similarity (the Jaccard coefficient for two sets A and B is defined as the ratio between the intersection $A \cap B$ and the union $A \cup B$).

As far as the threshold is concerned, in order to merge two segments SCAN requires a (reasonably) high similarity, not necessarily close to one. In fact, two segments might deserve being merged, even though their similarity is not extremely high. Suppose one of the two segments is substantially larger, containing a higher number of terms than the second segment, and let us further assume it contains the second segment as a sub-segment. In this situation, their similarity may be arbitrarily low. However, the segmentation approach [9], [10] protects us against such a situation, as the computed segments are ensured to be cohesive and decoupled (see Section III). In fact, the algorithm has no reward in gluing together non-cohesive methods. In essence, our aim is to merge segments that share no less than 50% of the linguistic information. By trial-and-error, we found that reasonable threshold values range between 60% and 80%, where 80% might become too conservative (i.e., it might not merge segments that are different just because of a different thread interleaving).

Once pairs of corresponding segments have been identified, the merger generates a synthetic trace. Continuing with the example above, a synthetic trace is generated containing n segments. Each segment is the result of a (possibly multiple) union (i.e., the union of the corresponding segment method signatures): $s_i \cup z_j$, for all pairs where $\sigma(z_i, s_j)$ is above threshold. The merger keeps track of the pairs of merged segments, so as to be able to map the information computed in subsequent phases to the original segments.

B. Relevant Term Identification

This step represents the core of the proposed approach, and aims at labeling segments. The first issue to consider is to choose the most appropriate source of information. Since execution traces are composed of method invocations, we could consider (i) terms contained in method signatures only, (ii) the whole source code lexicon of the invoked methods, or (iii) as before but including comments. Since a previous study [28] found that lexicon from method signatures provide more meaningful terms when labeling software artifacts, and since often developers tend to pay more attention when labeling API rather than when naming local variables or commenting source code, we decided to use only terms contained in the signature of invoked methods and their parameters.

Given a split trace T and a segmentation $S = (s_1, \ldots, s_n)$, SCAN extracts the signatures of all the invoked methods for each identified segment s_i. Then, SCAN models the segmentation as a set of documents, where each segment s_i is a document, and computes for each term $t_l \in s_i$ the *tf-idf* metric [23]. Specifically, *tf-idf* provides a measure of the relevance of a term for the segment, rewarding terms having a high-frequency in a segment (high *tf*) and appear in few segments (high *idf*). We make the hypothesis that a term appearing often in a particular segment, but not in other segments, provides linguistic information important for that given segment. Previous work [28] aimed at empirically investigating automatic labeling strategies for static source code artifacts suggest the use of *tf-idf*.

SCAN ranks the terms of the segment terms by *tf-idf* and keeps the topmost ones. The number of retained terms is supposed to be a compromise between a synthetic and a verbose description. Several possible strategies are foreseeable to select the top-ranked terms. First, it is possible to retain a maximum percentage (say top 10%) of the terms that have the highest ranking; second, a gap-based strategy is applicable (*i.e.*, retaining all terms up to when the difference between two subsequent terms in the ranked list is above a certain percentage gap); and third, one could choose a fixed number of topmost terms. In this paper, we adopt the latter strategy, and we found that considering the topmost 10 to 20 terms represents a reasonable compromise, that can produce meaningful segment labels.

C. Identifying Concepts Using Formal Concept Analysis

While the labeling produced in the previous step is expected to fully describe a segment functionality, what is still missing is the information about relations existing between different segments. For example, segments with identical reduced term description may appear multiple times, in different trace areas. Furthermore, two segments may share many terms, which possibly indicates the existence of a higher level concept common to both segments.

To discover this kind of information, SCAN uses FCA to highlight commonalities and differences between segments, by identifying terms shared between multiple segments and terms that are specific to particular segments. Specifically, SCAN produces a FCA context, where objects are segments and attributes are segment relevant terms. The produced lattice is then manually inspected by the software engineer.

Figure 1 shows an example of an FCA lattice for the ArgoUML scenario "add a new class". We can notice some relations between segments. For example, Segments 4, 10 and 16 are linguistically identical; they actually implement the same feature. They are also in relation with Segments 2, 8 and 14, as they share with those segments the term "display". Segments 4, 10 and 16 belong to the same concept, while being in different parts of the input trace. Methods of Segment 4 are called before those of Segment 10. Moreover, there are other segments between Segment 4 and Segment 10. In practice, it can be noticed that Segments 4, 10 and 16 represent the same computational phase repeated in different trace areas. This is represented very clearly and explicitly in the concept lattice of Figure 1. The same holds for segments sharing the same super-concept (e.g., "display"), which indicates a common functionality executed in different phases.

V. Case Study

The *goal* of this study is to evaluate SCAN, with the *purpose* of assessing its capabilities to label segments and to identify relations between such segments. The *quality focus* is the comprehension of execution traces. Maintainers have to perform this task during program understanding. The *context* consists of execution traces collected from two Java systems, JHotDraw and ArgoUML.

JHotDraw[3] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. *JHotDraw* has been widely used in various research works due to its structure (based on extensive adoption of design patterns) and documentation. In this evaluation we have used *JHotDraw* release 5.1, which consists of 155 classes in about 8 KLOC. *ArgoUML*[4] is an open-source UML modeling tool with advanced features, such as reverse engineering and

[3] http://www.jhotdraw.org
[4] http://argouml.tigris.org

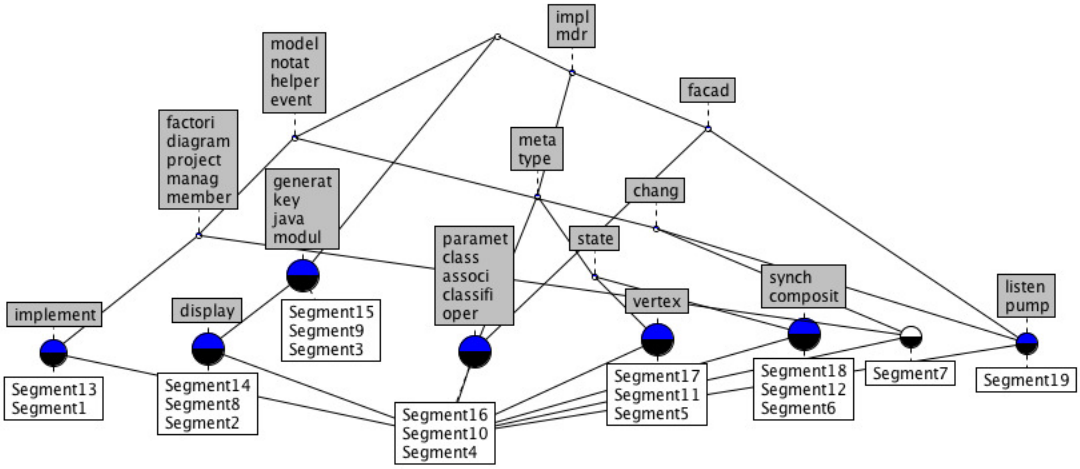

Figure 1. ArgoUML FCA lattice for the scenario "add a new class".

code generation. The ArgoUML project started in September 2000 and is still active. Similarly to *JHotDraw*, *ArgoUML* has been widely studied and used in various research works. We have used *ArgoUML* release 0.19.8, which contains 1,230 classes in about 113 KLOC.

For both programs, we collected execution traces for different scenarios. Specifically, we reused some of the scenarios previously used to validate trace segmentation [9], [10], plus we added some more, based on the knowledge we gained about this application. Tables I and IV reports details about the exercised scenarios and the collected traces. In the following, we refer to each scenario with a brief English sentence such as "Draw Ellipse, Delete Ellipse". We imply that, when the scenario is executed, other than the two features (drawing an ellipse and deleting it), also application start-up and shut-down are executed.

The study aims at answering the following two research questions:

- **RQ1.** *How effective is SCAN in assigning labels to segments?*
- **RQ2.** *Does SCAN help to discover relations between segments? Does it help to discover the macro phases in a trace?*

To address **RQ1**, one of the authors manually built labels for each segment and validated the SCAN results. We then compare manually built labels with the ones produced by SCAN by computing precision and recall [29] for each segment i of a scenario j:

$$Precision_{i,j} = \frac{|M_{i,j} \cap S_{i,j}|}{|S_{i,j}|} \quad Recall_{i,j} = \frac{|M_{i,j} \cap S_{i,j}|}{|M_{i,j}|}$$

where $M_{i,j}$ is the set of words contained in the manually generated label for segment i of scenario j and, similarly, $S_{i,j}$ is the set of words produced by SCAN. Note that, before computing precision and recall, we preprocess the manual labels similarly to what done when producing labels automatically. Specifically, (i) we split compound words (using camel case and underscore heuristics), (ii) we remove English stop words, and (iii) we perform Porter stemming.

Table I
STATISTICS OF JHOTDRAW COLLECTED TRACES.

Systems	Scenarios	Original Size	Compressed Sizes	Number of Segments
JHotDraw	Draw Rectangle (1)	15,706	930	54
	Draw Rectangle (2)	4,850	555	35
	Draw Rectangle, Delete Rectangle (1)	5,960	554	32
	Draw Rectangle, Delete Rectangle (2)	5,960	554	32
	Draw Ellipse (1)	4,545	556	36
	Draw Ellipse (2)	5,252	562	33
	Draw Ellipse, Delete Ellipse (1)	10,760	953	53
	Draw Ellipse, Delete Ellipse (2)	17,931	1,433	74
	Draw Rectangle, Draw Ellipse (1)	10,908	864	23
	Draw Rectangle, Draw Ellipse (2)	17,471	1,096	46
	Draw Rectangle, Draw Ellipse (3)	8,790	690	30

To address **RQ2**, we analyze the lattice produced by FCA to identify relations between different segments.

In the following, we report results aimed at addressing the three research questions, presenting, for the sake of clarity, all results for JHotDraw first, and then all results for ArgoUML.

A. JHotDraw

Table II shows SCAN generated labels in the first column and the manual labels in the second column for one of the JHotDraw scenarios. The top part of Table III reports descriptive statistics (first and third quartile, median, mean and standard deviation) of precision and recall. It can be noticed that, for instance, the mean Precision varies be-

tween 0.56 of "Draw Rectangle, Draw Eclipse" and 0.65 of "Draw Rectangle, Delete Rectangle", while the mean recall is stable around 0.81-0.82. Hence, for JHotDraw the automatic labeling performs relatively well, also considering that such results are perfectly in line with performances of automatically labeling of source code artifacts [28], which we argue are easier to label than execution traces.

To better understand the rationale of the identified segments and check the meaningfulness of the provided labels, we performed a fine-grained analysis of the segments. By exploring the content of each segment of the trace described in Table II, we found, for example, that Segment 1 contains methods that start the application (menu and icons creation). Segment 2 to Segment 24 correspond to phases needed to prepare canvas for creating and adding a figure to it. Furthermore, Segment 2, Segment 4 and Segment 24 contain methods to execute the "draw figure" command. Differently from the others, Segment 19 contains methods involved in bringing the selected figure to front and to send the other figures to back. Segment 20 contains methods needed to create box and figure locations. For segments between 24 and 33, we found that each of these segments corresponds to deletion and removal of figures, change listeners and events.

Similar results have been obtained for the other scenarios. In summary, we can claim that SCAN is able to assign labels in most of cases similar or representative to manually defined labels and that these labels actually correspond to the concepts encountered in the segments based on our manual inspection of code, documentation and executions.

To address **RQ2**, we exploited FCA to identify linguistically overlapping segments. In other words, segments having the same or shared labels implement similar or related concepts. By looking at Figure 2 we can notice that, for example, segments 4 and 23 are identical and implement the same concept. This was confirmed by manual inspection of the source code. A developer can therefore use lattice information to infer relations between segments and identify segments implementing the same feature/concept. We can also notice that sometimes a computation phase, represented as an FCA concept, is contained in a more abstract one. For example, in Figure 2 segments 28 and 30 are contained in a superconcept of the concept containing segments 26 and 31. In fact, they all share some labels (*listen, change, remove, figure*), but the latter segments (26, 31) have their own specific labels (*intern, multicast*).

B. ArgoUML

Table IV reports information about traces and identified segments for ArgoUML. As for JHotDraw, we compared the automatically generated labels with the ones produced manually. Table V shows, for the ArgoUML scenario "New Class", automatic and manually generated labels for the identified trace segments. For ArgoUML, the performance analysis of the comparison between manually produced

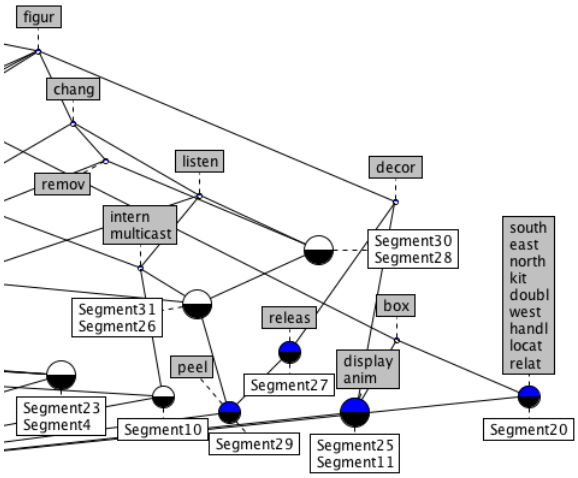

Figure 2. Excerpt of the JHotDraw FCA lattice for the scenario 'Draw Rectangle, Delete Rectangle".

Table IV
STATISTICS OF THE ARGOUML COLLECTED TRACES.

Systems	Scenarios	Original Size	Compressed Sizes	Number of Segments
ArgoUML	New Class (1)	82,579	2,785	22
	New Class (2)	60,853	2,239	19
	New Package(1)	13,115	800	15
	New Package (2)	21,423	1,642	19
	New Class, New Package (1)	38,940	1,220	13
	New Class, New Package (2)	50,650	1,146	13
	New Class, New Package (3)	36,408	1,251	12

labels and labels produced by SCAN (reported in the bottom part of Table III) reveals that performances are relatively lower than those obtained for JHotDraw. In particular, the mean precision ranges between 0.36 of "New Package" and 0.40 of "New Class", while the mean recall ranges between 0.48 of "New Class, New Package" and 0.64 of "New Class". The lower performances can be explained by the ArgoUML lexicon which is not as good as the JHotDraw one (JHotDraw was designed for pedagogical purposes, i.e., to show the usage of design patterns, hence source code artifacts are carefully named).

We also performed a sanity check of SCAN's capability of recognizing different instances of the same scenario. Figure 3 shows the FCA lattice produced for two executions of the scenario "New Package", with two trace instances, "NewPackage1" and "NewPackage2". All segments and concepts are similar between the two traces, except for segment 10 of "NewPackage1", which confirms SCAN's ability to recognize the occurrence of the same concepts in different executions.

We performed an in-depth analysis by exploring the

Table II
SCAN GENERATED AND MANUAL LABELS FOR THE JHOTDRAW TRACE "DRAW RECTANGLE, DELETE RECTANGLE".

Segment Number	Automatic Label	Manual Labels
1	draw iconkit creat palett text tool button line imag icon	Create drawing palette button tool and create icons kit.
2	draw cut transfer figur command view	Execute draw figure command.
3	draw menu copi shortcut past add command transfer duplic view	Add a command with the given shortcut to the menu.
4	draw transfer delet figur command view	Execute draw figure command.
5	ungroup draw group command view	Command to group and ungroup the selection into a group figure.
6	draw back send bring command front view	Create a command to bring to front and send to back the selected figures from others.
7	applic draw creat menu align command	Application menu creation and draw command.
8	applic draw graphic java palett menu button paint command tool	Draw command and get the selected botton from the menu palette tool.
9	add figur chang listen	Add a figure change listener.
10	add figur multicast intern chang event listen	Add a figure change event.
11	box decor anim display figur	Display the box and the borders of the figure.
12	figur set initi attribut	Initialize figure attributes.
13	figur empti size	Verify if the figure size is empty.
14	draw tool view editor standard	Draw the standard drawing tool.
15	draw applic tool set button	Set the tool of the editor.
16	graphic execut button revers enumer paint command tool select view	Execute command to paint the selected graphic button.
17	delet command execut duplic	Execute command to delete the duplicated selection.
18	ungroup group command execut	Execute command to group and ungroup the selection into a group figure.
19	execut back send bring command front	Execute command to bring to front and send to back the selected figures from others.
20	box relat locat handl west doubl kit north east south	Handle the locations and display the box of the figure.
21	unlock view unfreez draw standard	Unfreezes the view by releasing the drawing lock.
22	draw tool standard key press event view	Handling the key events in the drawing view.
23	draw transfer delet figur command view	Execute draw figure command.
24	draw remov request standard delet figur chang bounc event select	Delete the event from the selection.
25	box decor anim display figur	Display the box and the borders of the figure.
26	remov intern figur multicast chang event listen	Remove the figure change event.
27	releas decor figur	Release the figure decorator.
28	remov figur chang listen	Remove the figure change listener.
29	decor peel remov intern figur multicast releas chang event listen	Remove the figure change listener.
30	remov figur chang listen	Remove the figure change listener.
31	remov intern figur multicast chang event listen	Remove the figure change event.
32	draw enabl execut standard command key element check select view	Execute command to check enabled elements key from the selection.

Table III
DESCRIPTIVE STATISTICS OF PRECISION AND RECALL WHEN COMPARING SCAN LABELS WITH MANUALLY-PRODUCED LABELS.

JHotDraw										
Scenario	Precision					Recall				
	Q1	median	Q3	mean	σ	Q1	median	Q3	mean	σ
Draw Rectangle	0.50	0.60	0.83	0.64	0.25	0.75	0.83	1.00	0.81	0.20
Draw Rectangle, Delete Rectangle	0.50	0.60	0.72	0.65	0.21	0.70	0.80	1.00	0.82	0.15
Draw Rectangle, Draw Eclipse	0.40	0.60	0.70	0.56	0.22	0.67	0.80	1.00	0.81	0.19
ArgoUML										
Scenario	Precision					Recall				
	Q1	median	Q3	mean	σ	Q1	median	Q3	mean	σ
New Class	0.29	0.40	0.50	0.40	0.13	0.50	0.67	0.75	0.64	0.14
New Package	0.29	0.33	0.50	0.36	0.17	0.50	0.50	0.71	0.54	0.21
New Class, New Package	0.20	0.33	0.50	0.38	0.24	0.25	0.50	0.67	0.48	0.20

content of each segment of the traces. By manually inspecting code and documentation of ArgoUML, as well as the Cookbook for Developers [30], we found that Segment 1 contains methods for system start-up: Setup the project and implement factory and helper interfaces that control the lifetime and properties of elements in the repository. Segment 2 to Segment 7 correspond to "prepare creation" and "addition" of a new UML Class. For example, Segment 2 and Segment 3 contain methods to generate the module identification key. Segment 4 contains methods to create a class and define parameters. Similar results have been obtained for the other scenarios.

Figure 1 shows the FCA lattice for the execution trace of the scenario "New Class". As for JHotDraw, also for ArgoUML FCA helps to highlight relations between segments. For example, segments 4, 10 and 16 implement the same concept. The concept containing segments 3, 9 and 15 is a super-concept of the one containing segments 2, 8 and 14 and in fact it points to higher level concepts (*generate key java module*), while the sub-concept includes segments specific of the *display* functionality.

To identify macro phases in a trace, we consider relations between cohesive sets of segments, regarded as execution phases. One phase is built by repeated segments in a trace.

Table V
SCAN GENERATED AND MANUAL LABELS FOR THE ARGOUML TRACE "NEW CLASS".

Segment Number	Automatic Label	Manual Labels
1	event member helper notat manag project diagram implement model factori	Add project member and implement factory and helper interfaces
2	modul java display key generat	Display the module identification key
3	modul java key generat	Generate the module identification key
4	oper type facad classifi meta mdr associ class impl paramet	Create class and define parameters
5	vertex state meta mdr type impl	Display the state vertex
6	composit state meta synch mdr type impl	Display SynchSTate and composite state
7	member helper notat factori project diagram chang model event manag	Manage diagram events changes
8	modul java display key generat	Display the module identification key
9	modul java key generat	Generate the module identification key
10	oper type facad classifi meta mdr associ class impl paramet	Create class and define parameters
11	vertex state meta mdr type impl	Display the state vertex
12	composit state meta synch mdr type impl	Display SynchSTate and composite state
13	event member helper notat manag project diagram implement model factori	Add project member, implement factory and helper interfaces
14	modul java display key generat	Display the module identification key
15	modul java key generat	Generate the module identification key
16	oper type facad classifi meta mdr associ class impl paramet	Create class and define parameters
17	vertex state meta mdr type impl	Display the state vertex
18	composit state meta synch mdr type impl	Display SynchSTate and composite state
19	notat helper facad event pump mdr model chang impl listen	Add model event change listener

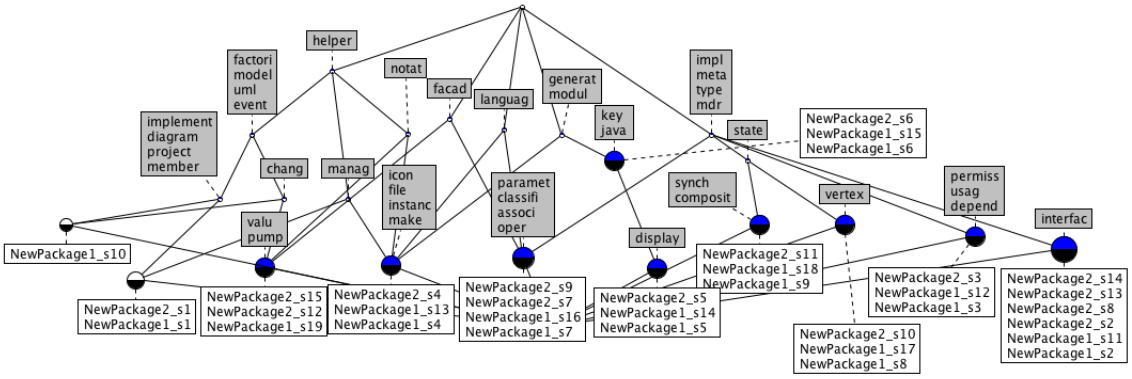

Figure 3. FCA lattice for the scenario "New Package".

For example, in Figure 1, segments 2, 3, 4, 5 and 6 define an execution phase on the trace and this phase is repeated two times: first with segments 8, 9, 10, 11 and 12, and then with segments 14, 15, 16, 17 and 18. The rest of the segments are also converted to an execution phases.

After defining the phases we can draw a higher level flow diagram of phases with labels as shown in Figure 4, using the temporal relations between phases. The "New Class" scenario, generating 32 segments, can be summarized into four macro execution phases. The first phase deals with the system startup, this is followed by activity needed to create class and properties (*e.g.*, state, composite, etc). The third phase is devoted to managing diagram events and, finally, the last phase models *add events* and *model changes*.

While phase recognition is currently a manual process, we plan to investigate methods to automate it, as part of our future work. Since the FCA lattice combined with the temporal ordering of segments was very useful for manual phase recognition, we think that there is ample room for automation of this process.

C. Discussion

For what concerns **RQ1**, quantitative results might be read as indicators of poor performance of the label assignment algorithm, with the recall/precision around 50% and above. As mentioned above, the achieved performance are in line with those obtained when comparing automatically generated and manually generated labels for source code artifacts [28]; moreover, the obtained results confirm that also for execution traces a simple labeling based on lexicon extracted from method signature is enough.

Also, if we complement the quantitative data with the qualitative investigation performed on the automatically labelled segments, we can conclude that this level of similarity between automatic and manual label sets is definitely adequate to support program understanding tasks. This is because we expect that developer with some knowledge about the application would find it relatively easy to distill the relevant concepts from the automatic labels, even if such labels contain some noise and overlap only by 50% with the manually produced labels. For instance, consider the label

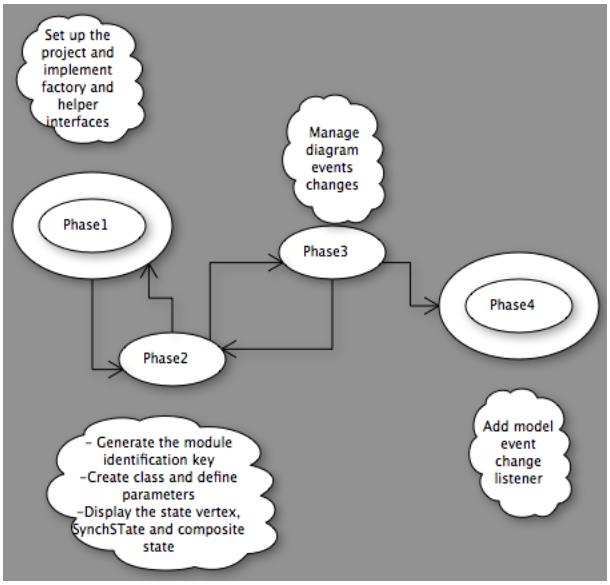

Figure 4. Flow diagram of phases for the scenario "New Class".

set produced for Segment 1 of JHotDraw (see Table II): it is relatively easy for someone having a (even limited) application knowledge to recognize the terms *creat draw palett button tool iconkit* as key terms for the implemented concept. Even though the manually produced label is longer and more explanatory (*Create drawing palette button tool and create icons kit*), the terms selected from the automatically produced label represent a very good and crisp summary of it. Similar considerations can be applied for Segment 18 of ArgoUML, where *synch composite state* are a meaningful summary for *Display SynchState and composite state*.

Regarding the inter-concept relations and the manual recognition of phases (**RQ2**), qualitative results indicate that the automatically-produced labels, organized into a concept lattice where similar or identical segments are grouped together, are extremely useful to understand commonalities and differences between segments and to extract a view where macro phases can be labelled by the terms associated with the super-concepts of the involved segments. Cohesive sets of similar segments can be identified in the concept lattice. Such sets, in turn, define macro phases, that labelled with super-concept terms. The temporal ordering of the segments involved in different macro-phases suggests the temporal organization of the recognized phases. We think this has huge potential in supporting comprehension of complex execution scenarios for large software systems.

D. Threats to Validity

Threats to *internal validity* concern confounding factors that could affect our results. These could be due to the presence, in the execution traces, of extra method invocations related to GUI events or other system events. Also, the order of invocation in different executions may depend on multi-threading. This may affect *tf-idf* values and could produce different results in terms of relevant information. The frequency-based pruning and the analysis of different execution trace instances for one scenario mitigate these threats.

Threats to *external validity* concern the possibility to generalize our results. Although we apply our approach on traces from two different systems, further studies on larger traces and more complex systems are needed, especially to better demonstrate accuracy in assigning labels representative of concepts implemented by trace segments.

VI. Conclusion

This paper proposed SCAN, an approach aimed at supporting developers to discover concepts in segments of execution traces by (i) assigning labels (sets of words) to each segment, (ii) discovering relations between segments via formal concept analysis, and (iii) helping to group segments into macro phases. SCAN has been conceived on top of the trace segmentation approach presented in [9], [10] in mind. However, it is not tied to any specific trace segmentation approach.

To evaluate the accuracy and effectiveness of SCAN in assigning meaningful sets of words representative of the concepts implemented in segments, we have performed a manual validation on several traces of both JHotDraw and ArgoUML, two widely-known Java applications, often used as a benchmark in software engineering research.

We performed both a qualitative and a quantitative validation aiming at verifying the relation between manually defined labels and segment labels automatically generated by SCAN. Quantitative analysis shows different ranges of similarities between manual and automatic labels. On JHotDraw the median of recall (precision) 80% (60%), while for ArgoUML it is above 50% (30%). We believe that for the given task it is important to favor recall over precision and developers are able to quickly discard wrong information while retrieving the information is a long and expensive task.

The manual inspection of automatically-produced labels indicates that these are quite informative and useful to reconstruct the target concepts associated with each segments. The relatively low precision values should not be interpreted as poor performance. On the contrary, our qualitative analysis indicates that such performance is sufficient for manual concept assignment and phase recognition, and it is perfectly in line with performances of automatic labeling of source code artifacts using information retrieval methods [28]. Finally, the lattices obtained by FCA help to highlight meaningful relations among segments and to successfully highlight phases of execution scenarios.

While phase recognition is still mostly manual work, we plan to investigate how to automate it in our future work. Other directions for future work will also focus on further validation of SCAN with a pool of independent

developers. We also intend to study the effect of object-oriented naming conventions, and to further study the impact of multi-threading in label assignment. Moreover, we would like to investigate the possibility of applying SCAN to label multiple trace segmentations, i.e., segmentations of traces corresponding to different scenarios.

REFERENCES

[1] V. Kozaczynski, J. Q. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1065–1075, 1992.

[2] T. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the International Conference on Software Engineering*, 1993, pp. 482–498.

[3] N. Anquetil and T. Lethbridge, "Extracting concepts from file names: a new file clustering criterion," in *Proceedings of the International Conference on Software Engineering*. IEEE CS Press, 1998, pp. 84–93.

[4] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance - Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[5] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *Proceedings of Working Conference on Reverse Engineering*, 2004, pp. 112–121.

[6] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification: An epidemiological metaphor," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 627–641, 2006.

[7] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[8] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proceedings of the International Conference on Program Comprehension*. IEEE CS Press, 2008, pp. 53–62.

[9] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A heuristic-based approach to identify concepts in execution traces," in *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE CS Press, 2010, pp. 31–40.

[10] S. Medini, P. Galinier, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "A fast algorithm to locate concepts in execution traces," in *Proceedings of the International Symposium on Search-based Software Engineering*. IEEE CS Press, 2011, pp. 252–266.

[11] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of Working Conference on Reverse Engineering*. IEEE CS Press, 2010, pp. 35–44.

[12] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2010, pp. 43–52.

[13] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceeding of the International Conference on Software Engineering*. ACM, 2011, pp. 101–110.

[14] ——, "Generating parameter comments and integrating with method summaries," in *International Conference on Program Comprehension*. IEEE CS Press, 2011, pp. 71–80.

[15] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, and A. Mehrabian, "The concept of stratified sampling of execution traces," in *Proceedings of the International Conference on Program Comprehension*. IEEE CS Press, 2011, pp. 225–226.

[16] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah, "Exploiting text mining techniques in the analysis of execution traces," in *Proceedings of the International Conference on Software Maintenance*. IEEE CS Press, 2011, pp. 223–232.

[17] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[18] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, March 2003.

[19] P. Tonella, "Concept analysis for module restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, 2001.

[20] T. Paolo and A. Giuliano, "Object-oriented design pattern inference," in *Proceedings of the International Conference on Software Maintenance*, 1999, pp. 230–238.

[21] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[22] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[23] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[24] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[25] N. Ali, Y. Gueheneuc, and G. Antoniol, "Requirements traceability for object oriented systems by partitioning source code," in *Proceedings of the Working Conference on Reverse Engineering*, 2011, pp. 45 –54.

[26] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of the International Conference on Software Maintenance*. IEEE CS Press, 2006, pp. 469–478.

[27] B. G. R. Ville, *Formal Concept Analysis*. Springer: Mathematical Foundations, 1999.

[28] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?" in *Proceedings of the International Conference on Program Comprehension*. IEEE, 2012.

[29] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[30] E. Tolke, M. Klink, L. Tolke, and M. Van Der Wulp, "Cookbook for developers of argouml: an introduction to developing argouml," 2004.