JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **00**:1–35
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

SCAN: An Approach to Label and Relate Execution Trace Segments

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

[1] *DGIGL, École Polytechnique de Montréal, 2900, Boul. Édouard-Montpetit, Montréal, Québec, H3T 1J4, Canada*
[2] *Department of Engineering, University of Sannio, Via Traiano 1, 82100 Benevento, Italy*
[3] *Fondazione Bruno Kessler (FBK), Via Sommarive 18, 38050 Povo, Trento, Italy*

SUMMARY

Program comprehension is a prerequisite to any maintenance and evolution task. In particular, when performing feature location, developers perform program comprehension by abstracting software features and identifying the links between high-level abstractions (features) and program elements.
We present SCAN (Segment Concept AssigNer), an approach to support developers in feature location. SCAN uses a search-based approach to split execution traces into cohesive segments. Then, it labels the segments with relevant keywords and, finally, uses formal concept analysis to identify relations among segments. In a first study, we evaluate the performances of SCAN on six Java programs by 31 participants. We report an average precision of 69% and a recall of 63% when comparing the manual and automatic labels and a precision of 63% regarding the relations among segments identified by SCAN. After that, we evaluate the usefulness of SCAN for the purpose of feature location on two Java programs. We provide evidence that SCAN 1) identifies 69% of the gold set methods and 2) is effective in reducing the quantity of information that developers must process to locate features—reducing the number of methods to understand by an average of 43% compared to the entire execution traces.

Received ...

KEY WORDS: Concept identification; dynamic analysis; information retrieval; formal concept analysis.

1. INTRODUCTION

Program understanding activities are preliminary to any software maintenance and evolution task. Among these activities, feature location is the activity during which developers identify the program elements contributing to implement specific domain concepts, program features, or computation phases, as well as understand the relations among these elements [1, 2]. The understanding that developers gain allows them to accomplish their tasks, *e.g.*, fix a bug or enhance an existing feature.

Among other works several researchers proposed approaches to ease feature location, *e.g.*, [1, 2, 3, 4, 5]. These approaches typically use static and–or dynamic information extracted from the source code of a program or from some execution traces to relate method calls to user-visible features. These approaches also use several different techniques to locate feature in source code and–or execution traces, *e.g.*, Antoniol *et al.* [4] proposed an epidemiological metaphor to analyze

[1]Correspondence to: DGIGL, École Polytechnique de Montréal, Pavillon Mackay et Lassonde, 2900, Boul. Édouard-Montpetit, Montréal, Québec, H3T 1J4, Canada. E-mail: venera.arnaoudova@polymtl.ca

source code, Poshyvanyk *et al.* [6] used latent-semantic indexing (LSI) and then a combination of formal-concept analysis and LSI [7] to locate feature from source code and execution traces, Rohatgi *et al.* [5] used graph dependency ranking on static and dynamic data, Pirzadeh *et al.* [8] studied psychology laws describing how the human brain groups similar methods in execution traces.

Cornelissen *et al.* [9] present a systematic survey of 176 articles from the last decade on program comprehension through dynamic analysis. They found the first article on program comprehension through dynamic analysis back to 1972 where Biermann builds finite state machines from execution traces [10]. Despite the advantages of dynamic analysis techniques, there are also known drawbacks, one of which is *scalability* [9]: “The scalability of dynamic analysis due to the large amounts of data that may be introduced in dynamic analysis, affecting performance, storage, and the cognitive load humans can deal with.” Indeed execution traces are a precious source of information but they can be overly large and noisy. For example, the trace corresponding to the simple scenario “Draw a rectangle” in JHotDraw v5.1 contains almost 3,000 method invocation-related events. Hence, execution traces might not be of immediate support to developers for program comprehension activities, in general, and feature location, in particular. To address scalability issues, some approaches propose to compact execution traces (*e.g.*, Reiss and Renieris [11], Hamou-Lhadj and Lethbridge [12]), build high-level behavioral models (*e.g.*, Hamou-Lhadj *et al.* [13], Safyallah and Sartipi [14], Lo *et al.* [15] [16]), extract dynamic slices (*e.g.*, Agrawal *et al.* [17], Zhang *et al.* [18]), and segment execution traces (*e.g.*, Asadi *et al.* [19], Medini *et al.* [20], Pirzadeh and Hamou-Lhadj [8]).

Approaches analyzing execution traces, *e.g.*, [6, 8], generally aim at mapping user-visible features to segments in traces. However, when splitting an execution traces, not all segments correspond to the user-activated features. Other segments exist to support the user-activated features, for example segments containing the method calls necessary to set up the user-interface or segments with methods performing background save of opened files. Therefore, it is often not sufficient to segment traces: segment must be labeled as well and related to one another within a trace and across traces. Paraphrasing Eisenbarth *et al.* [21]: “Assigning meaning to the [segments] was generally simple for [segments] [...] specific to a subset of all features. Understanding the [segments] became increasingly difficult when [more generic segment] were inspected.”

We specifically tackle the problem of splitting traces into segments, assigning labels to segments, and relating these segments in and across traces to help developers understand execution traces. We propose SCAN, Segment Concept AssigNer, an approach to support feature location by breaking execution traces into segments and labeling those segments to represent the features that they execute. SCAN starts from a set of execution traces obtained by exercising a program of interest under different scenarios. It uses dynamic programming [22, 23] to identify cohesive segments in the execution traces, as described in our previous work [19, 20]. Additionally, to cope with multi-threading, SCAN merges similar segments from different execution traces: segments of method calls whose methods have high conceptual cohesion [24] and low coupling with methods in other segments. Finally, SCAN uses Information Retrieval (IR) techniques to identify sets of keywords, called labels, describing the segments as well as Formal Concept Analysis (FCA) to identify the relations among segments: (1) segments activating the same feature, (2) segment(s) activating part of a feature activated by other segment(s), and (3) sequences of segments activating the same set of features. Thus, SCAN helps developers to understand the features implemented by these segments.

This paper extends our previous work [25], which proposed a preliminary approach to label segments and identify relations among them. The novel contributions of this paper are:

1. An approach to reduce segment size by retaining only the most important method calls using *tf-idf* and Vector Space Model (VSM) IR techniques. In our previous work, we considered all method calls as being equals when splitting the traces and labeling their segments;
2. An algorithm to automatically identify relations among segments using FCA and lattice partitioning. In our previous work [25], this identification was performed manually by partitioning concept lattices produced by FCA;

3. An empirical evaluation involving 31 participants and six programs of (1) the impact of segment size reduction on the labels, (2) the accuracy of the labels generated by SCAN, and (3) the accuracy of the relations among segments reported by SCAN;
4. An empirical study of the usefulness of SCAN using two other real-world programs and their execution traces and bug reports. We provide evidence that SCAN is effective in retrieving relevant methods and reducing the quantity of information that developers must process to locate features.

Outline. This paper is organized as follows. Section 2.1 provides the necessary background. Section 3 describes SCAN. Section 4 describes the empirical evaluation of the performances of SCAN. Section 5 describes and reports the results of the empirical study of the usefulness of SCAN. Section 6 discusses threats to the validity of the performances and usefulness studies. Section 7 relates this work to the existing literature on trace segmentation, feature location, and code summarization. Finally, Section 8 concludes the paper and outlines directions for future work.

2. BACKGROUND

In this section, we briefly summarize key concepts and techniques used in this paper. Specifically, we summarize our approach for trace segmentation and we report a primer on formal concept analysis (FCA). We use FCA to discover commonalities between segments and high level relations between segments.

2.1. Trace Segmentation

This section summarizes the trace segmentation approach by Medini *et al.* [20], which aims at grouping together subsequent method calls that form conceptually cohesive segments [24]. To collect execution traces of the program under study we execute a scenario and we use the MoDeC instrumentor to record the sequence of events exercised during such execution. MoDeC is part of the Ptidej Tool suite and is used to extract and model sequence diagrams [26]. Before segmenting the traces, SCAN filters out *utility* methods repeated in various trace regions, *e.g.*, methods related to mouse events, by analyzing the distributions of method calls. Then, it compresses the trace, using a run-length encoding algorithm, to remove repetitions of method calls. From the filtered and compressed trace, SCAN collects unique methods and models each method as a document, by collecting the method name, parameters, and body from the program source code. To this end, SCAN parses the source code and creates an Abstract Syntax Tree (AST) using the Eclipse Java Development Tools (JDT). SCAN processes each document (method) by applying the Latent Semantic Indexing (LSI) technique as usually applied in software engineering [24]. It first extracts terms from the source code, splits compound identifiers separated by camel case (*e.g.*, `getBook` is split into `get` and `book`), removes programming-language keywords and English stop words, and performs stemming [27]. SCAN then indexes the obtained terms using the *tf-idf* weighting scheme [28]. It thus obtains a term–document matrix on which it applies LSI to reduce the size of the term–document matrix [29]. In the current implementation of SCAN, we choose, as in previous work [19, 20], an LSI subspace size equal to 50. We use FacTrace to generate the term–document matrix [30].

Finally, SCAN uses a dynamic-programming optimization technique to segment the trace using the term–document matrix. The cost function driving the optimization relies on conceptual cohesion and coupling measures [24, 31] rather than structural cohesion and coupling measures. SCAN computes the quality of the segmentation of a trace split into K segments using the fitness function (fit) defined in Equation 1, which balances between the cohesion of a segment (the higher the better) and its coupling with all other segments in the trace (the lower the better).

$$fit(segmentation) = \frac{1}{K} \times \sum_{i=1}^{K} \frac{COH_i}{COU_i + 1} \tag{1}$$

$$COH_l = \frac{\sum_{i=begin(l)}^{end(l)-1} \sum_{j=i+1}^{end(l)} \sigma(m_i, m_j)}{(end(l) - begin(l) + 1) \times \frac{(end(l)-begin(l))}{2}} \tag{2}$$

$$COU_l = \frac{\sum_{i=begin(l)}^{end(l)} \sum_{j=1, j<begin(l) \text{ } or \text{ } j>end(l)}^{N} \sigma(m_i, m_j)}{(N - (end(l) - begin(l) + 1)) \times (end(l) - begin(l) + 1)} \tag{3}$$

SCAN computes the cohesion of a segment l (denoted as COH_l) as the average (textual) similarity between the source code of all pairs of methods called in l. Equation 2 shows the definition of COH_l in which $begin(l)$ is the position of the first method call of segment l and $end(l)$ the position of the last method call in l. The similarity $\sigma(m_i, m_j)$ between two methods m_i and m_j is computed using the cosine similarity measure over the term–document matrix.

SCAN computes the coupling of segment l (denoted as COU_l) as the average similarity between methods m_i belonging to segment l and methods m_j outside l. It uses the formula shown in Equation 3 in which N is the length of the trace. COU_l represents the average similarity between methods in l and those in the rest of the segments.

More details can be found in previous work by Asadi *et al.* [19] and Medini *et al.* [20].

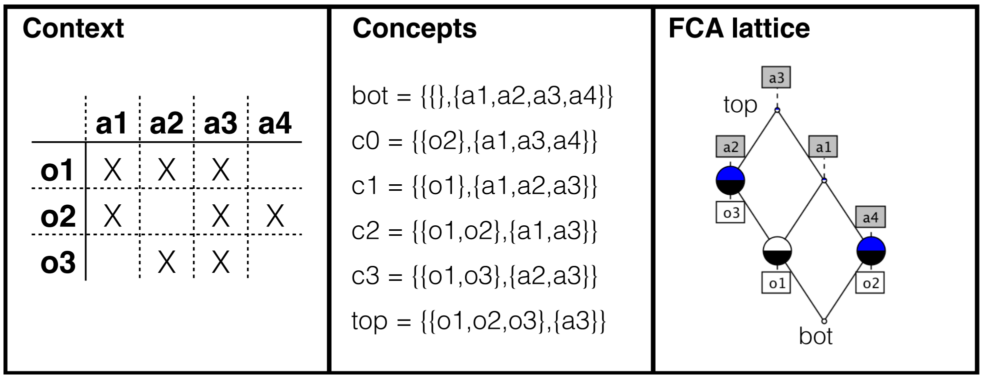
2.2. *Formal Concept Analysis*

This section provides the details of Formal Concept Analysis (FCA) technique and the algorithm used to build a FCA lattice. FCA [32] groups *objects* that have common *attributes*. The starting point for FCA is a *context* (O, A, P), *i.e.*, a set of objects O, a set of attributes A, and a binary relation among objects and attributes P, stating which attributes are possessed by which objects. A *FCA concept* is a maximal collection of objects that have common attributes, *i.e.*, a grouping of all the objects that share a set of attributes. More formally, for a set of objects $X \subseteq O$, a set of attributes $Y \subseteq A$, and the binary relation between them P, a FCA concept is the pair (X, Y) such that:

$$X \mapsto X' = \{a \in A \mid \forall o \in X : (o, a) \in P\}$$
$$Y \mapsto Y' = \{o \in O \mid \forall a \in Y : (o, a) \in P\}$$

where $X' = Y$ and $Y' = X$. X' is the set of attributes common to all objects in X and Y' is the set of objects possessing all attributes in Y. X is the *extent* of the concept and Y is its *intent*. The extent X of a concept is obtained by collecting all objects reachable from (X, Y) down to the bottom node. The intent Y is obtained following the opposite direction, *i.e.*, from (X, Y) to the top node, and by collecting all reachable attributes. The concepts define a lattice.

To build a FCA lattice from a set of segments and terms, we use the general bottom-up algorithm described by Siff and Reps [33]. First, the algorithm computes the bottom element of the concept lattice. Next, it computes atomic concepts. Atomic concepts are the smallest concepts with an extent containing each object treated as a singleton set, such as $c0$ in Figure 1. Then, the algorithm closes the set of atomic concepts under join. Initially, a work-list is formed containing all pairs of atomic concepts (c', c) where $c \nsubseteq c'$ and $c' \nsubseteq c$. While the work-list is not empty, the algorithm removes the element $(c0, c1)$ from the work-list and computes $c'' = c0 \cup c1$. If c'' is a concept that is not yet discovered, then it adds all pairs of concepts (c'', c) to the work-list, where $c \nsubseteq c''$ and $c'' \nsubseteq c$. This process is repeated until the work-list is empty.



Context

	a1	a2	a3	a4
o1	X	X	X	
o2	X		X	X
o3		X	X	

Concepts

bot = {{},{a1,a2,a3,a4}}
c0 = {{o2},{a1,a3,a4}}
c1 = {{o1},{a1,a2,a3}}
c2 = {{o1,o2},{a1,a3}}
c3 = {{o1,o3},{a2,a3}}
top = {{o1,o2,o3},{a3}}

Figure 1. Concepts partition example.

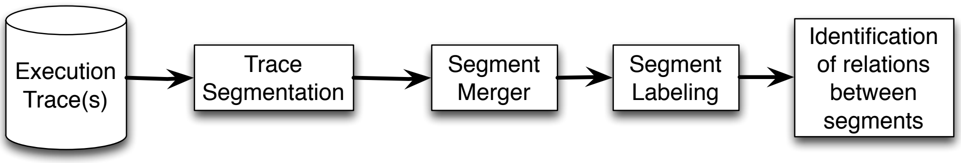


Figure 2. Overview of SCAN.

3. THE SCAN APPROACH

SCAN accepts as input one or more execution traces obtained by exercising some scenarios in a program. Such execution traces can be obtained by executing the scenarios for which the developer is interested to perform feature location. Then, as depicted in Figure 2, it consists in a series of four steps. In *Step 1*, SCAN splits traces into cohesive segments, using the approach previously proposed by Medini *et al.* [20]. Then, in *Step 2*, it merges similar segments using the Jaccard measure on terms extracted from the segments. After that, in *Step 3*, it uses an IR-based approach to label segments. Finally, in *Step 4*, it uses FCA to identify relations among segments.

3.1. Definitions

In the following, we use a vocabulary similar to that used in previous works. An execution *trace* is a sequence of events occurring during the execution of a scenario. Execution traces are generally very large and the occurring events, or *method calls*, likely relate to multiple features of the program. An execution trace can be divided into segments, where each *segment* is a cohesive sequence of method calls activating one feature or a small set of features. A segment is a set of successive method calls, which is characterized by the number of method calls and the number of distinct (unique) methods called. A *label* can be assigned to each segment, *i.e.*, a set of words describing the possible feature(s) activated by the segment. Two or more segments activating the same feature are part of the same *phase*. Finally, repeated sequences of phases can be abstracted into *macro-phases*. A macro-phase thus activates a set of features.

3.2. Step 2: Segment Merging

In this step, SCAN merges similar segments using the Jaccard measure on terms extracted from the segments. The aim of the merging is to recognize *similarities* among segments belonging to multiple execution traces and merge these segments. Indeed, we expect to have a great number of common segments between multiple execution traces of a same scenario or, even, of different scenarios even though the ordering of the segments and some other segments may be different among traces due to thread interleaving, variations in application inputs (some of which cannot be fully controlled while instrumenting the programs to collect traces), or variations in machine-load conditions. We thus suppose that highly similar segments in different traces contribute to the same feature, regardless of the specific thread interleaving or trace region in which they occur. SCAN merges segments from different execution traces of a same scenario only. It handles the relations among segments in a same trace in Step 4, using Formal Concept Analysis, as described in Section 3.4.

Let $S = (s_1, \ldots, s_n)$ and $Z = (z_1, \ldots, z_m)$ be two segmentations of two traces, *i.e.*, two sequences of segments. For each s_i, SCAN computes all similarities $\sigma(s_i, z_j)$ and keeps pairs above a given threshold. The *similarity* between two segments is computed as the Jaccard coefficient between the segments terms, extracted from the term–document matrices of the segments. The Jaccard coefficient for two sets A and B is defined as:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The higher the number of terms in common between two segments, the higher the similarity. Once SCAN has identified all pairs of similar segments, it generates a *synthetic trace* containing n segments. Each segment is the result of a (possibly multiple) union of s_i and z_j where $\sigma(s_i, z_j)$ is above the threshold. SCAN attempts to find both one-to-one as well as one-to-many relations among the segments. It keeps track of the pairs of merged segments to allow mapping the information computed in subsequent steps back to the original segments.

Consider for example a trace Z split into four segments $Z = (z_1, z_2, z_3, z_4)$ and a threshold equal to 70%. First, SCAN computes the similarities between s_1 and the segments of the trace Z as follows $\sigma(s_1, z_1) = 0.8$, $\sigma(s_1, z_2) = 0.2$, $\sigma(s_1, z_3) = 0.6$, and $\sigma(s_1, z_4) = 0.9$. Then SCAN generates a synthetic segment s'_1 by merging the segments s_1, z_1, and z_4 since $\sigma(s_1, z_1)$ and $\sigma(s_1, z_4)$ are greater than 70%.

SCAN expects a (reasonably) high similarity between segments to merge two segments but this similarity is not necessarily close to one. Indeed, two segments might deserve to be merged even though their similarity is lower: let us suppose that one of the two segments is contained in the second segment as a sub-segment. In this situation, the segment similarity may not be very high. Therefore, the threshold should also not be very high. Using a lower threshold does not compromise the accuracy of the merging because the computed segments are ensured to be cohesive and decoupled, as explained in the previous Section 2.1 and, consequently, the algorithm has no incentive in putting together non-cohesive methods.

We explain in details our experimental choice of the threshold used by SCAN in Section 4.2.

3.3. *Step 3: Label Identification*

In this step, SCAN uses an IR-based approach to label segments. The first issue when labeling segments is the choice of the most appropriate source of information. Segments are composed of method calls and, thus, we could consider (1) the terms[2] contained in method signatures only, (2) the whole source code lexicon of the called methods, or (3) the whole source code lexicon and related comments. However, a previous study [34] reported that lexicon from method signatures provide more meaningful terms when labeling software artifacts than other sources. Moreover, developers often pay more attention to method signature when understanding source code. Consequently, we decided to use only terms contained in the signatures of the called methods. Given a trace segmentation $S = (s_1, \ldots, s_n)$, SCAN extracts the signatures of all the called methods in each identified segment s_i. Then, it models the segments as a set of documents, uses Vector Space Model (VSM) to represent segments as vectors of terms and computes for each term $t_l \in s_i$ the *tf-idf* weighting scheme [28]. Specifically, *tf-idf* provides a measure of the relevance of the terms for a segment, rewarding terms having a high term frequency in a segment (high *tf*) and appearing in few segments (high *idf*). We make the hypothesis that a term appearing often in a particular segment but not in other segments provides important information for that segment.

Then, SCAN ranks the terms in the segment by their *tf-idf* values and keeps the top ones. The number of retained terms is a compromise between a succinct and a verbose description. Several

[2] We assume that method signatures are not obfuscated.

possible strategies exist to select the top terms. First, SCAN could retain a maximum percentage, *e.g.*, top 10%, of the terms that have the highest ranking; second, SCAN could apply a gap-based strategy, *i.e.*, retain all terms up to when the difference between two subsequent terms in the ranked list is above a certain percentage; and, third, SCAN could choose a fixed number of top terms. The retained terms form the label of a segment.

We explain in details the choice of the strategy used by SCAN in Section 4.2.

3.4. Step 4: Relation Identification

While we expect the labels produced in the previous step to fully describe the feature implemented by a segment, they do not help developers to relate segments in a same trace with one another. For example, segments with identical labels may appear multiple times, in different trace regions. Furthermore, two segments may share many terms, which could possibly indicate the existence of a higher-level feature common to both segments. To discover such relations among segments, SCAN uses Formal Concept Analysis (FCA) and highlights commonalities and differences among segments by identifying terms shared between multiple segment labels and terms that are specific to particular segment labels. In SCAN, objects are segments and attributes are the terms in the segment labels. The binary relation states which term is included in which segment label. A FCA concept is a cohesive set of segments sharing some terms in their labels. Figure 3 shows an example of a FCA lattice for the ArgoUML scenario "add a new class" in which each node represents a formal concept (X, Y). To visualize the lattice we use Concept Explorer[3] [35]. Blue (black) filled upper (lower) semi-circle indicates that an attribute (object) is attached to the concept. SCAN uses the lattice to identify relations among segments as explained in the following.

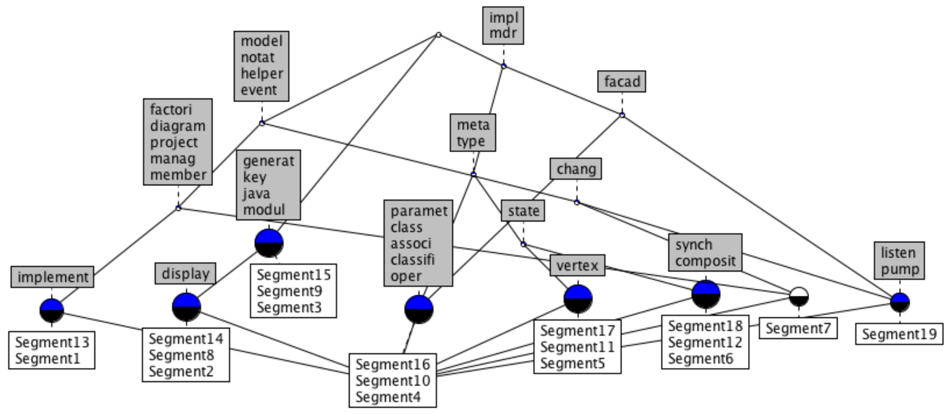


Figure 3. ArgoUML FCA lattice for the scenario "add a new class".

Types of Relations. By applying FCA on the segments and terms from their labels and analyzing the resulting lattices, we identified the following relations among segments: same phase, sub/super feature, and macro-phase.

Two distinct segments sharing the same relevant terms are considered to activate the same feature, thus forming a phase. For example, in Figure 3, SCAN identifies Segments 2, 8, and 14 as part of the *same phase* because these segments belong to the same concept. These three segments share the same terms and actually activate the same feature.

Sub-feature relation exists when a set of segment(s) activate part of a feature of another set of segment(s). SCAN identifies a sub-feature relations between two segments when relevant terms in the label of one segment are a superset of the terms of the label of another segment, *i.e.*, by selecting

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

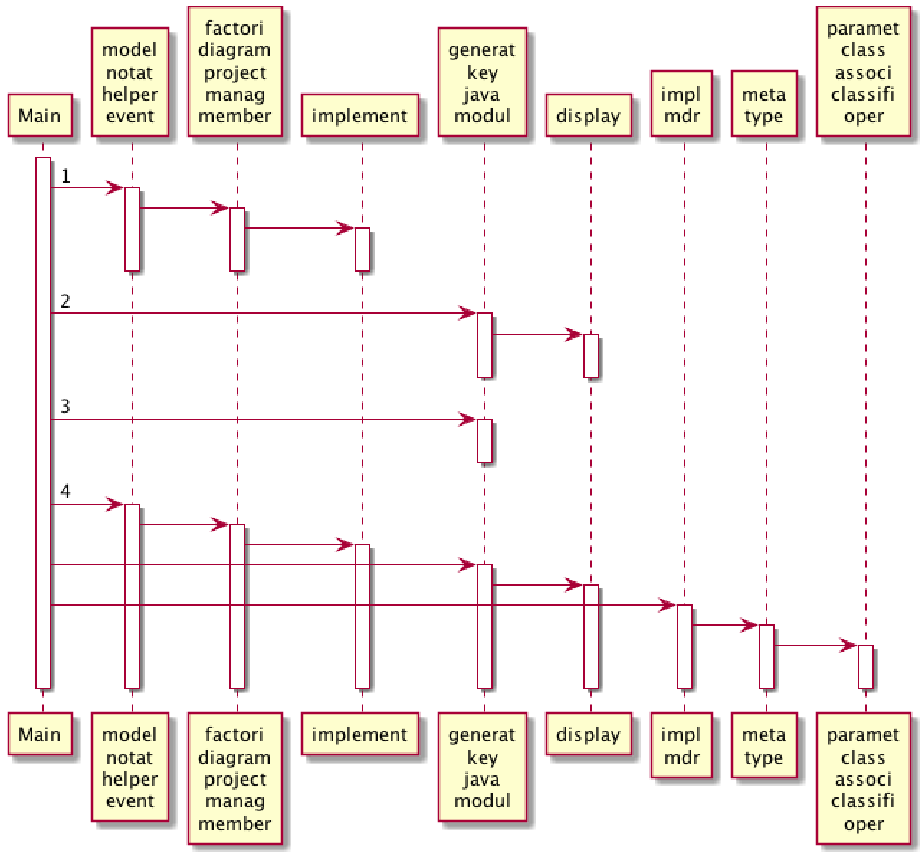


Figure 4. ArgoUML sequence diagram derived from the FCA lattice for the scenario "add a new class".

the intent of a concept of interest in the lattice. For example, in Figure 3, Segments 2, 8, and 14 share the terms "*generat, key, java, and modul*" with the Segments 3, 9, and 15 and thus are a sub-feature of these segments. Conversely, a *super-feature* relation exists when terms in the label of one segment are a subset of the terms of another.

A *macro-phase* is the result of the abstraction of repeated sequences of identical phases, which activates a set of features. SCAN identifies macro-phases by finding repeating sequences of FCA concepts. For example, in Figure 3, there are several phases such as: Phase 2: Segments 2, 8, and 14; Phase 3: Segments 3, 9, and 15; Phase 4: Segments 4, 10, and 16; Phase 5: Segments 5, 11, and 17; and Phase 6: Segments 6, 12, and 18. A segment activates the phase that it belongs to, thus, for example, Segment 2 activates Phase 2. The next segment in the trace, Segment 3, activates Phase 3. In the same way segments 4, 5, and 6 activate respectively Phases 4, 5, and 6. Thus, the sequence of Segments 2 to 6 activates the sequence of Phases 2 to 6. However, the same sequence of phases is also activated with the sequence of Segments 8 to 12 as well as with the sequence of Segments 14 to 18. Thus, the three sequences of Segments 2 to 6, Segments 8 to 12, and Segments 14 to 18, activate the same features, *i.e.*, activate the features of Phases 2 to 6. SCAN abstracts those five phases and identifies the macro-phase containing Phases 2 to 6.

Example of Sequence Diagram. The FCA lattice shown in Figure 3 can be used by developers in the more familiar form of a UML sequence diagram. To obtain a sequence diagram from the FCA lattice, segments are considered in the order in which they appear in the execution trace. Each segment is associated with its most specific FCA concepts in the FCA lattice. Methods are activated in the sequence diagram for each FCA attribute of the segment-specific FCA concepts. The topmost reachable FCA attributes are activated first and all FCA attributes in the sub-FCA concepts are activated as nested operations.

A portion of the sequence diagram for the FCA lattice in Figure 3 is shown in Figure 4 (generated using PlantUML[4]). The sequence diagram shown in Figure 4 contains the same information available from the FCA lattice, but the sequential ordering of the called method makes it easier to read and understand for developers.

Segment 1 is associated with a FCA concept that has three super-FCA concepts, two of which are annotated with FCA concept-specific attributes. Starting from the topmost annotated FCA concept, the following methods are activated in the sequence diagram: *"model, notat, ..."*, *"factori, diagram, .."*, and *"implement"*. Similarly, Segment 2 activates *"generate, key, ..."*, which has a nested activation labeled *"display"*, while Segment 3 activates only *"generate, key, ..."*. For the sake of clarity, only a portion of the method calls in Segment 4 are shown.

4. EVALUATING THE PERFORMANCES OF SCAN

The *goal* of the experimental evaluation is to evaluate the performances of SCAN with the *purpose* of assessing its capability to (1) select the most important methods of a segment, (2) label segments, and (3) identify relations among segments. The *quality focus* is the comprehension of execution traces. The *perspective* is of researchers interested in providing support to program comprehension by labeling and relating segments in execution traces.

4.1. Research questions

The evaluation aims at answering the following research questions:

- **RQ1:** *How do the labels of the trace segments produced by the participants change when providing them different amount of information?* This research question investigates whether providing participants with the list of the most relevant methods in a segment is sufficient to produce labels. We rank methods by relevance according to their *tf-idf* [28], *i.e.*, methods frequently invoked in the particular segment but not in other segments.
- **RQ2:** *How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN?* This research question evaluates the performances of SCAN when labeling segments. Similarly to De Lucia *et al.* [34] when evaluating the labeling of software artifacts, we aggregate labels produced by the participants and compare the sets of most frequent terms with the labels automatically produced by SCAN.
- **RQ3:** *To what extent does SCAN correctly identify relations among segments?* This research question assess the use of FCA by scan to identify relations among segments. We ask participants to assess the relations among segments identified by SCAN.

4.2. Study Set Up

This section describes the set up of the experimental evaluation. It describes the research questions that we aims to answer, the *objects*, *i.e.*, execution traces of six Java programs, the *participants*, *i.e.*, 31 students and professionals that participated in the experiment, and the choice of the thresholds and strategy for the trace segmentation and the label identification.

4.2.1. Objects

The objects of our evaluation are execution traces collected from six Java programs belonging to different domains. ArgoUML[5] is a leading open-source UML modelling tool. It supports all standard UML v1.4 diagrams. JHotDraw[6] is a Java GUI framework for technical and structured graphics. Its

[4]http://plantuml.sourceforge.net/
[5]http://argouml.tigris.org/
[6]http://www.jhotdraw.org/

Table I. Program characteristics.

Programs	LOCs	Trace	Trace size
ArgoUML v0.19.8	163K	New class new package	36K
JHotDraw v5.1	8K	Draw rectangle delete rectangle	6K
Mars v4.3	32K	Screen magnifier	673K
Maze r186	9K	Micro mouse	1,075K
Neuroph v2.1.0	10K	Kohonen visualizer	75K
Pooka v2.0	44K	New account new e-mail	36K
		Create folder open folder	23K
Total for the 6 systems		7 traces	

Table II. Segments Characteristics.

Programs - Traces	# of segments	Number of method calls					
		Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
ArgoUML - New class new package	12	2	2	2	104	37	714
JHotDraw - Draw rectangle delete rectangle	32	2	2	2	17	3	183
Mars - Screen magnifier	30	2	2	2	11	3	167
Maze - Micro mouse	75	2	2	2	30	3	999
Neuroph - Kohonen visualizer	4	2	2	4	8	10	23
Pooka - New account new e-mail	60	2	2	2	33	3	1,038
Pooka - Create folder open folder	18	2	2	2	78	5	957
Overall	231	2	2	2	34	3	1,038

design relies heavily on some well-known object-oriented design patterns because it was developed as an example of the use of design patterns. Mars[7] is a simulator for the MIPS assembly language. It also includes a lightweight interactive development environment (IDE) for programming in MIPS. Maze[8] is a micro-mouse maze editor and simulator. It provides statistics on the comparisons of different mazes and AI algorithms. Neuroph[9] is a lightweight Java neural network framework to develop common neural network architectures. It includes a library of neural network concepts and a user-interface to create, train, and save networks. Pooka[1] is an email client written in Java, using Swing and JavaMail. It supports IMAP, POP3, and Unix-style mailbox folders. It also has support for encryption using PGP and S/MIME.

The choice of these six programs was of convenience, driven by the availability of documentation to ease the participants' task of labeling the segments and by the possibility to generate traces related to different execution scenarios. Table I summarizes characteristics of the programs, *i.e.*, their sizes (in terms of lines of code), short descriptions of the scenarios used to generate the execution traces, and the sizes of the traces (in terms of number of executed events, *i.e.*, constructor and method calls). We used one scenario per program, except for Pooka, for which we used two scenarios. Table II reports descriptive statistics about the numbers and sizes of the segments that SCAN identifies in the execution traces.

4.2.2. Participants

The experiment involved a total of 31 participants. Eight of them were professionals, *i.e.*, developers, researchers, or postdoctoral research fellows, and the others were students, *i.e.*, Ph.D., M.Sc., or B.Sc. Table III provides descriptive statistics of the participants' programming experience. All participants were volunteers.

[7]http://courses.missouristate.edu/kenvollmar/mars/index.htm
[8]http://code.google.com/p/maze-solver/
[9]http://neuroph.sourceforge.net/index.html
[1]http://www.suberic.net/pooka/

Table III. Participants characteristics.

	# of Participants	Years of programming experience		
		Min.	Median	Max.
Students	23	3	7	14
Professionals	8	4	9	14
Overall	31	3	7	14

None of the participants is an original developer of the object programs. This lack of knowledge could decrease the participants' ability to properly comprehend the traces. However, developers of large software programs may not be familiar with the entire program and thus would have been subject to the same threat. To cope with this threat, we ask more than one participant to perform the same task. Because none of the participants know the programs, we do not have to take into account possible differences between "novices" or "experts" that could influence our results [36].

4.2.3. *Trace Segmentation Threshold*

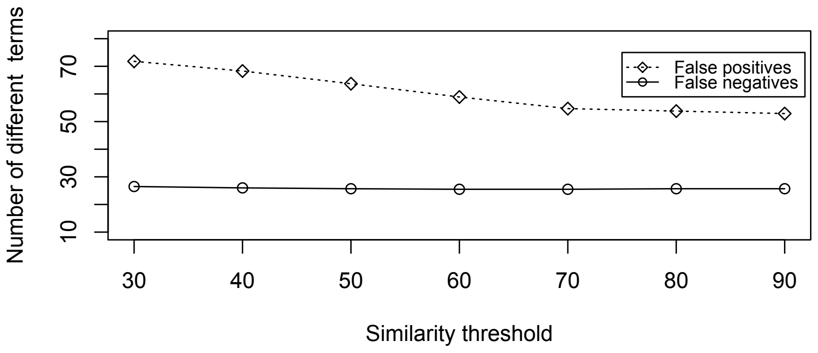


Figure 5. False positives and negatives according to different threshold values.

To determine a threshold providing a good accuracy when SCAN merges segments, we performed an experiment using five scenarios for ArgoUML and JHotDraw and their 11 corresponding execution traces. We varied the threshold values from 30% to 90% and evaluated the accuracy of the merged segments using the labels that SCAN generates for these merged segments and the manual labels. Our hypothesis is that the variation of the false positives and negatives is due to the merging of segments. Thus, when segments that pertain to the same feature are merged the number of false positives and negatives will be lower compared to when segments pertaining to different features are merged. The solid line in Figure 5 shows the average number of terms in the manual labels that did not appear in the automatic labels, *i.e.*, false negatives. The average is between 26.5 for a threshold at 30% and 25.7 at 90%. The minimum value is equal to 25.5, at 60% and 70%. The values are very close thus the threshold does not significantly affect the merged segments in terms of false negatives. The dotted line in Figure 5 shows the average number of terms in the automatic labels that did not appear in the manual labels, *i.e.*, false positives. The average is between 71.8 for a threshold at 30% and 52.9 at 90%. Using 30% as a similarity threshold, the number of different terms is more important because SCAN merges segments that pertain to different features. False positive values are more stable between 70% and 90%, between 54.7 and 52.9. Thus, we choose the value 70%, in the range of 70% to 90%.

Table IV. Segments used to evaluate the filtering using *tf-idf*.

Programs	Traces	Segments IDs	Full segments sizes	# of unique methods
ArgoUML	New class new package	s5	92	49
JHotDraw	Draw rectangle delete rectangle	s1	183	77
		s20	69	41
Mars	Screen magnifier	s1	167	160
		s22	93	82
Maze	Micro mouse	s4	142	71
		s45	102	36
Pooka	New account new e-mail	s52	131	91
	Create folder open folder	s1	88	67
Overall	9 segments		1,067	674

4.2.4. *Label Identification Strategy*

SCAN ranks the terms in segments by their *tf-idf* values and keeps the topmost ranked terms. The number of retained terms must be a compromise between a succinct and a verbose description. Several possible strategies are foreseeable to select the top-ranked terms. First, it is possible to retain a maximum percentage (e.g., 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 of 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 terms represents a reasonable compromise, which yields meaningful segment labels. Note that this value represents the maximum number of terms, thus, segments containing very few methods may be labeled by SCAN with fewer terms.

4.3. *Experimental Design and Analysis*

In the following, we describe the evaluation design and procedure followed to answer the three research questions.

4.3.1. *RQ1: How do the labels of the trace segments produced by the participants change when providing them different amount of information?*

When the size of a segment (in terms of its numbers of method calls) is large, it is difficult to understand. To reduce the time and effort for understanding a segment, we propose to characterize a segment using only the calls to 5 or 15 different, unique methods. Note that a method can be called more than once in a segment. We selected the values in a way to have one small and one medium versions of the segment (5 and 15 respectively). The small version reduces the number of methods to understand substantially but may result in loss of relevant information. The medium version of the segment is likely to better preserve the relevant information but at the expense of the larger number of methods that one must understand.

To address **RQ1**, we compare the labels produced by the participants when showing them three versions of a same segment: full, *i.e.*, the segment in its original size; medium, *i.e.*, a subset of the segment reduced to the calls to only 15 unique methods; and small, *i.e.*, a subset of the segment reduced to the calls to only 5 unique methods. We select the unique methods using the top-most ranked methods according to the *tf-idf* weighting scheme [28]. A segment subset is obtained by removing all method calls other than the top 5 or 15 from the original segment. The order of the method calls are preserved.

The experiment is designed as follows. We select nine segments (belonging to different programs) whose full sizes, *i.e.*, numbers of method calls, is between 50 and 200. We set an upper limit to control the participants' effort. We set a lower limit to ensure that the medium and small subsets of the segments are still meaningful and do not reduce only to a couple of methods. Table IV shows the segments used for this part of the experiment, their original sizes, and the numbers of unique methods.

We group participants into three groups, G1, G2, and G3. We assign each version of a segment to a different group. For example, to G1, we assign the subset of the top 5 unique method calls of segment *s5* of ArgoUML, "New class new package", representing a total of 12 method calls as some methods are called more than once. To G2, we assign the subset of the top 15 unique method calls of the same segment resulting in a total of 31 method calls. To G3, we give the segment in its original version (92 method calls to 49 unique methods). Participants belonging to each group label an equal number of small, medium, and full segments.

We evaluate the filtering approach, *i.e.*, the approach of reducing the size of segments, from two aspects: (1) the degree to which information is preserved and (2) the degree to which it preserves the agreement between participants, as explained in the following.

Preservation of Information. We use the labels produced by the participants working on the full segments as oracle to assess the preservation of information in the medium and small subsets of the segments. Thus, to evaluate the preservation of information, we compute the intersection between terms provided by participants working with medium and small segments and those produced with the full segments. The greater the intersection between the reduced (medium and small) and full versions of the segments, the higher the recall.

Preservation of Agreement among Participants. We consider the number of terms on which participants agree to be representative of a segment and of the degree of agreement among participants. Again, we use as oracle the labels produced by the participants working on full segments. To evaluate the preservation of agreement among participants, we use a two-way permutation test [37] to verify if the degree of agreement is significantly influenced by (1) the number of participants considered to compute the agreement, (2) the size of the segment subset given to the participants (*i.e.*, full segment, medium and small subsets), and (3) the interaction between the number of participants and the segment size. Thus, we investigate if (1) the agreement decreases when a larger number of participants provide labels (because different participants may provide different labels) and (2) the agreement changes when providing participants with a different amount of information, *i.e.*, full or reduced versions of the segments.

The permutation test [37] is a non-parametric alternative to the two-way ANalysis Of VAriance (ANOVA). Differently from ANOVA, it does not require data to be normally distributed. It builds the data distributions and compares the distributions by computing all possible values of the statistical test while rearranging the labels (representing the various factors being considered) of the data points. We used the implementation available in the *lmPerm* R package. We have set the number of iterations of the permutation test to 500,000. The permutation test does sample permutations of combinations of factor levels and, therefore, multiple runs of the test may produce different results. We made sure to choose a high number of iterations so that results did not vary over multiple executions of the test. When performing the test, we assume a significance level $\alpha = 0.05$.

To assess the agreement among participants, we follow a rule to decide when two terms are considered equivalent. In our previous work [20], we ruled that there exist an agreement between two participants on a term if both participants provide two terms sharing the same stem. In the following, we extend this rule to synonyms (*e.g.*, *shape* and *figure*), terms holding a hypernym/hyponym relation (*e.g.*, *display* and *screen*), and terms holding a holonym/meronym relation (*e.g.*, *point* and *location*). We use WordNet [38] to obtain these taxonomic relations among terms and we will refer to terms sharing those relations as *synsets*.

We considered different options to select the numbers of participants that must choose a synset to consider this synset representative of a segment. For example, we could consider a synset if at least one participant chooses it. Such a choice is equivalent to considering the union of all terms proposed by all participants. A minimum of two participants would mean that we consider only synsets chosen by at least two participants. The number of required participants can grow until it reaches the total number of participants, which would mean that a synset must be chosen by all

Table V. Examples of labels provided by SCAN and the participants.

	Labels	SCAN Precision	SCAN Recall
SCAN	figure, listener, add, internal, multicaster, event, change		
Participants - intersection	figure, event, change	43%	100%
Participants - union	trigger, figure, event, change, listener, multicaster, manage, add	86%	75%

Table VI. Example of relations among segments.

	Segments	Labels	Relations
SCAN	9	listener, add, change, figure	sub/super feature
	10	figure, listener, add, internal, multicaster, event, change	
Participant I	9	composite, figure, trigger, event	sub/super feature
	10	manage, figure, change, event, trigger	
Participant J	9	abstract, figure, change, add, listener	same feature
	10	figure, change, event, multicaster, add, listener	

participants to be considered representative. To avoid making choice that could bias the results of the experiment, we consider the entire range of possible values in our experiment.

4.3.2. *RQ2: How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN?*

To evaluate the labels produced automatically by SCAN, we first build an oracle consisting of 210 segments labeled manually by the participants. The inclusion criteria was that these segments must include less than 100 method calls to ease the participants' labeling tasks, because larger segments could have been more difficult for participants to understand. Each segment is labeled by at least one participant. More than half of the segments (116) are labeled by two participants. Then, we evaluate SCAN by computing the precision and recall of its automatically-generated labels with respect to the labels provided by the participants.

To explain how we performed the evaluation of the labels generated by SCAN, let us consider the automatic label produced by SCAN for a segment of JHotDraw and the corresponding manual labels provided by the participants, shown in Table V. The two possible operators to combine the manual labels are intersection and union. The first operator (*i.e.*, intersection) considers a synset to be relevant if both participants suggested it. The second operator (*i.e.*, union) considers a synset to be relevant if at least one of the participants suggested it. We observe that, depending on the operator, the precision and recall of SCAN varies. When the more conservative operator (intersection) is chosen, the number of synsets in the manual oracle significantly decreases thus resulting in higher recall but lower precision. Union provides a balance between the two measures. We show results for union and intersection.

To evaluate SCAN segment merging, we use two execution traces for each scenario presented in the Table IV of the five studied programs. We compute the precision and recall of the automatically-generated labels after merging the segments of the execution traces for the same scenario. We use the same oracle consisting of 210 segments labeled manually by the participants. Then, we compare the results of the labels produced automatically by SCAN with and without segment merging, *i.e.*, the results obtained using a single or multiple execution traces for the same scenario.

4.3.3. *RQ3: To what extent does SCAN correctly identify relations among segments?*

To address **RQ3**, we ask the participants to validate all relations among segments identified by SCAN. We do not ask participants to manually identify the relations among segments for two reasons. First, identifying the relations requires to compare each segment in a trace with any other segments, taking into account the possible reordering of method calls as well as inclusions. Thus, such a task would have been very demanding for the participants. Second, identifying these relations is not a task commonly undertaken by participants and, thus, its results would have been of a quality inferior to that of the labeling task, which participants perform implicitly or explicitly when understanding a segment. Participants validating the relations between segments use the full segments to understand and label the segments. Next, they use the labels that they just produced and the comprehension they have gained to validate the possible relations. We provide definitions for the different types of relations between segments: same feature (phase) or sub/super feature. For macro-phases, we ask the participants to validate all the phases of a given macro-phase. If all phases participating in a sequence of SCAN phases is validated by the participants then the macro-phase is also considered valid by construction.

We report the accuracy, *i.e.*, the percentage of relations that SCAN has correctly identified as vetted by the participants. We also report separately the accuracy for SCAN capability to identify super/sub-feature relations because of the participants' difficulty to distinguish sub-, super-, and same-feature as illustrated by the following example. Table VI shows two segments from JHotDraw labeled by SCAN and by two participants, Participant I and Participant J. Both SCAN and Participant I identify Segment 9 as activating a sub-feature of the feature activated by Segment 10. However, according to Participant J, the two segments activate the same feature. This example shows that distinguishing between sub/super feature and same feature is difficult. Therefore, when presenting the results, we report the percentages of agreements between the participants and SCAN with and without distinguishing between sup/super feature and same feature.

4.4. *Experiment Results*

This section reports the results of our experimental evaluation to address the research questions formulated in Section 4.1.

4.4.1. *RQ1: How do the labels of the trace segments produced by the participants change when providing them different amount of information?*

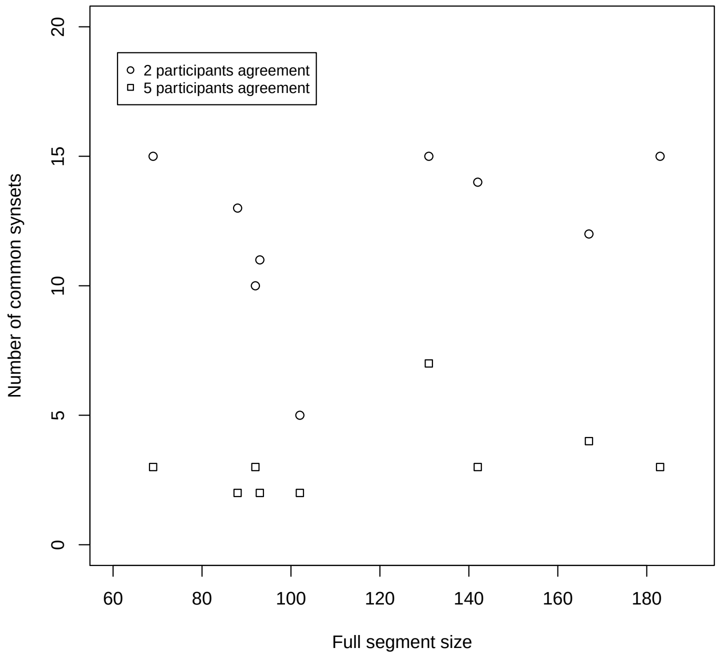


Figure 6. Agreement among participants for full segments.

Before analyzing the participants' labels with different subsets of the segments, we assess the quality of the participants' labels with full segments, *i.e.*, when using all information available. When the sizes of segments are large, the agreement among participants could be low due to the overwhelming amount of method calls, the complexity of the segments, or other factors, resulting in a random selection of terms. Figure 6 shows the number of synsets on which participants agree as a function of the sizes of the segments in their full version. To simplify the figure, we only show the cases of two and five participants. The figure shows that there is no linear relation between agreement and sizes, but rather a constant relation. This constant relation is confirmed by building a linear regression model and observing that the size of the segments is never a significant variable.

Table VII. Precision (P), Recall (R), and F-Measure (F) on the synsets of labels when comparing small and medium subsets versus full segments.

Segment	Small versus Full						Medium versus Full					
	2 participants			5 participants			2 participants			5 participants		
	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)
ArgoUML - New class new package: Segment 5	62	50	55	100	67	80	50	50	50	50	100	67
JHotDraw - Draw rectangle delete rectangle: Segment 1	62	33	43	0	0	0	73	53	61	20	33	25
JHotDraw - Draw rectangle delete rectangle: Segment 20	78	47	59	67	67	67	88	47	61	67	67	67
Mars - Screen magnifier: Segment 1	44	33	38	67	50	57	62	42	50	25	25	25
Mars - Screen magnifier: Segment 22	38	45	41	0	0	0	57	73	64	20	50	29
Maze - Micro mouse: Segment 4	30	21	25	40	67	50	60	43	50	67	67	67
Maze - Micro mouse: Segment 45	57	80	67	50	100	67	80	80	80	100	100	100
Pooka - New account new e-mail: Segment 52	64	60	62	50	14	22	82	60	69	67	29	40
Pooka - Create folder open folder: Segment 1	33	31	32	25	50	33	64	54	59	33	50	40
Overall	52	44	47	44	46	42	68	56	60	50	58	51

Preservation of Information. To analyze the amount of information lost when reducing the sizes of segments, we calculate the precision and recall of the labels produced with the reduced segments, *i.e.*, the small and medium subsets of the segments, with respect to the labels produced with the full segments. Table VII shows the results. We vary the number of participants considered for agreement and we show results for two and five participants[1].

We observe that, with the increase of the minimum number of participants, the variation of both precision and recall increases too. Thus, a smaller number of participants results in a smaller standard deviation of precision and recall across different segments. The larger standard deviation when increasing the number of participants is due to the small number of synsets in the manual labels of the segments. Thus, the size of the manual oracle decreases when the number of participants increases, which impacts negatively the evaluation of the precision of SCAN because the number of terms in the automatic labels is constant.

Considering the mean values of precision and recall and varying the numbers of participant between two and five, precision for small subsets varies between 42% and 52% and recall varies between 42% and 46%. For medium subsets of the segments, the mean values for precision and recall are greater and vary between 50% and 68% for precision and between 56% and 66% for recall. These results show that we can significantly reduce the number of methods that participants must use to understand a segment, *i.e.*, on average 92% for small and 76% for medium subsets, while keeping about half of the synsets that would appear if the segment was not reduced in size.

Considering small and medium subsets, we loose about half of the information, which explains the difference between labels produced using, on the one hand, medium and small segment, and, on the other hand, labels produced using the full segments. Participants analyzing reduced segments have less information and thus may tend to provide more details regarding the key concepts. However, participants understanding the full segments must provide more effort to extract the key concepts; they may produce more concise labels concerning the key concepts while trying to reach as many concepts as possible.

[1] When six participants is the minimum number required to consider a synset as representative for a segment, the resulting labels for some of the analyzed segments are empty.

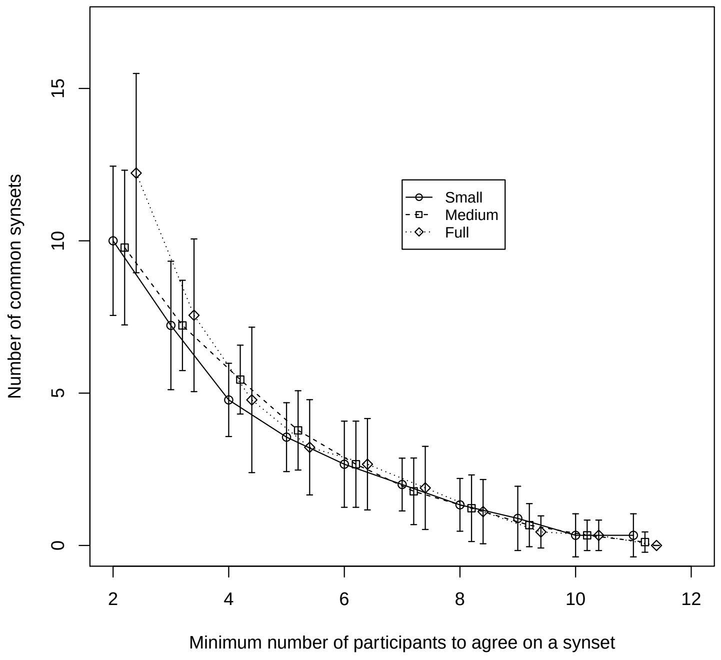


Figure 7. Agreement among participants.

Table VIII. Results of two-way permutation test of agreement by number of participants and size of the segment subset (full, medium, small).

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Participants	1	2448.7364	2448.7364	500000	<**0.0001**
Size	2	0.8222	0.4111	156267	0.8915
Participants:Size	2	11.3515	5.6758	500000	0.2062
Residuals	264	968.0566	3.6669		

Preservation of Agreement among Participants. Figure 7 shows the mean value and the standard deviation of agreements among participants for the nine segments, considering their full versions, as well as their reduced versions, *i.e.*, small and medium subsets. The only notable decrease in the number of synsets on which participants agree happens when two participants consider a synset as representative for a segment. We tested whether the agreement was influenced by (1) the number of participants, (2) the size of the provided segment subset (full, medium, small), and (3) their interactions. Results of the permutation test, shown in Table VIII, indicate that, while there is a significant difference of agreement when considering a different number of participants, as expected, the sizes of the subsets and their interactions with the number of participants do not significantly influence the agreement.

We thus answer **RQ1**: How do the labels of the trace segments produced by the participants change when providing them different amount of information? as follows: Small and medium subsets of segments preserve 50% or more of the synsets provided by participants while drastically reducing the amount of information that participants must process to understand a segment. The reduction of size with respect to the original size of the segment is on average 92% for small subsets and 76% for medium subsets.

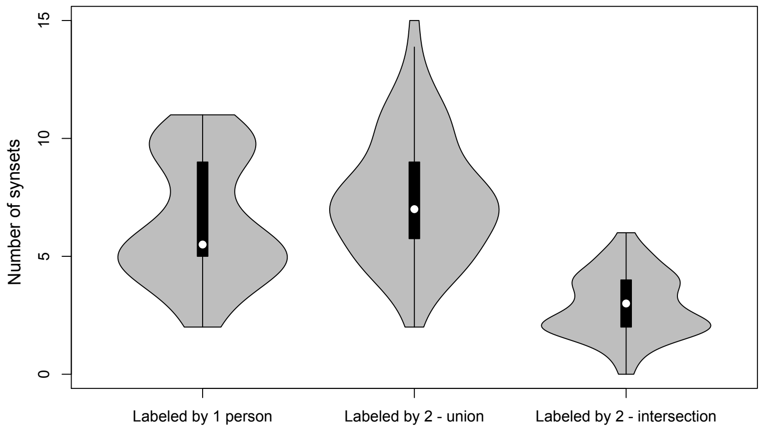


Figure 8. Number of synsets in manual labels.

Table IX. Precision (P) and Recall (R) of automatic labels assigned by SCAN without merging segments compared to oracle built by participants.

Program	1 participant only		2 participants - intersection		2 participants - union	
	P	R	P	R	P	R
ArgoUML	-	-	27%	50%	48%	37%
JHotDraw	-	-	53%	91%	82%	62%
Mars	60%	100%	48%	97%	65%	64%
Maze	60%	53%	18%	92%	32%	45%
Neuroph	28%	48%	-	-	-	-
Pooka	85%	82%	43%	93%	74%	73%
Averages	66%	62%	43%	90%	69%	63%

4.4.2. *RQ2: How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN?*

Figure 8 shows violin plots for the number of synsets in the manually-labeled segments. Violin plots [39] combine boxplots and kernel density functions, thus showing the shape of a distribution. The dot inside a violin plot represents the median; a thick line is drawn between the lower and upper quartiles; a thin line is drawn between the lower and upper tails. We observe from Figure 8 that segments that have been labeled by two participants have a median of 3 common synsets when considering intersection, but with a large proportion of values being concentrated at 2 synsets. We consider this number of common synsets to be a reasonable agreement provided that the median for segments that have been labeled by one participant only is at 5.5, again with high concentration in lower values—5. When considering the union of the synsets the median is higher at 7.

Table IX reports the results of evaluating the automatic labels when considering both operators. When considering all programs, the average values for precision and recall for segments labeled by one participant are 66% and 62%, respectively. For segments labeled by two participants, precision and recall values are 69% and 63% on average, respectively. Finally, when we consider the intersection of synsets for segments labeled by two participants, the recall is high, 90% on average, but precision can be as low as 18% with an average of 43%. This low precision is partially due to the lower number of synsets in the manually-labeled segments compared to the number of terms in the labels generated by SCAN. As shown in Figure 8, the median for the number of synsets in manual labels is three when intersection is considered. However, SCAN generates 10 terms for any label.

Table X shows examples of labels assigned by SCAN and the corresponding labels assigned by the participants. The first part of the table shows labels for which both precision and recall are low ($\leq 40\%$). The second part shows cases where SCAN has a precision and recall greater than 75%.

Table X. Examples of labels produced by SCAN and participants.

Examples of labels with low accuracy

Segment	SCAN label	Participants label
Maze - s9	info description cell template maze model size page painter icon	creating new maze setting its properties creating paths
Maze - s19	count controller step robot	model steps history move done turn
Neuroph - s1	network neural components tree project application neurons component easy folder view	initiate Kohonen view train randomize
Neuroph - s4	draw visualizer kohonen frame application neurons easy	return the view of the current Kohonen

Examples of labels with high accuracy

Segment	SCAN label	Participants label
Mars - s5	key dump stroke file memory action venus save icon	dump save file item menu action create memory
JHotDraw - s21	unlock standard unfreeze drawing view	unfreeze unlock standard view drawing
Pooka NewAccountNewMail - s6	network connection add listener item manager change	connection manager change item listener add network
Pooka CreateOpenFolder - s15	item connection network	network connection item id

Table XI. Precision (P) and Recall (R) of automatic labels assigned by SCAN after merging segments compared to oracle built by participants.

Program	1 participant only		2 participants - intersection		2 participants - union	
	P	R	P	R	P	R
ArgoUML	-	-	27%	50%	47%	36%
JHotDraw	-	-	53%	91%	82%	62%
Mars	60%	100%	48%	97%	65%	64%
Maze	59%	53%	18%	92%	32%	45%
Neuroph	27%	43%	-	-	-	-
Pooka	85%	82%	43%	92%	73%	71%
Averages	66%	62%	43%	90%	69%	63%

Table XII. Examples of relations detected by SCAN for Pooka, scenario “New account new e-mail”.

Phase Relations	**Segments in the Phase**	**SCAN Labels**	Participants
Phase 16	Segments 16, 28, and 41	load state wizard editor pane	√
Phase 17	Segments 17, 29, and 42	wizard state controller editor pane beginning	√
Phase 18	Segments 18, 30, and 43	set end wizard focus accept state editor pane property beginning	√
Phase 19	Segments 19, 31, and 44	controller wizard state pane editor	√
Phase 20	Segments 20, 32, and 45	composite focus accept label editor swing property	√
Macro-phases Relations	**Involved Phases**	**Number of Repetitions**	
Macro-phase 1	Phases 16, 17, 18, 19, and 20	The sequence of phases is repeated 3 times	√
Sub-feature Relations	**Segments/Phases involved**	**Details**	
Sub-feature 5	Phase 17 activates a sub-feature of Phase 19	Both activate features regarding state of the wizard editor pane. Phase 17 is specific to beginning state.	√
Sub-feature 10	Segment 38 activates a sub-feature of Segment 15	Both fire property committing events. Segment 38 fires additional property events.	√

Table XI reports the results of evaluating the automatic labels after merging the segments of two execution traces for each scenario when considering both operators. When considering all programs results are very close to the results obtained when using a single execution trace. The only notable exception is Neuroph, for which when we consider one participant only the recall drops with 5% (*i.e.*, from 48% for labels generated using a single execution trace to 43% for labels generated after the merge of segments of two execution traces of the same scenario).

We thus answer **RQ2**: How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN? as follows: SCAN automatically assigns labels with an average precision and recall of 69% and 63%, respectively, when compared to manual labels produced by merging the labels of two participants using union.

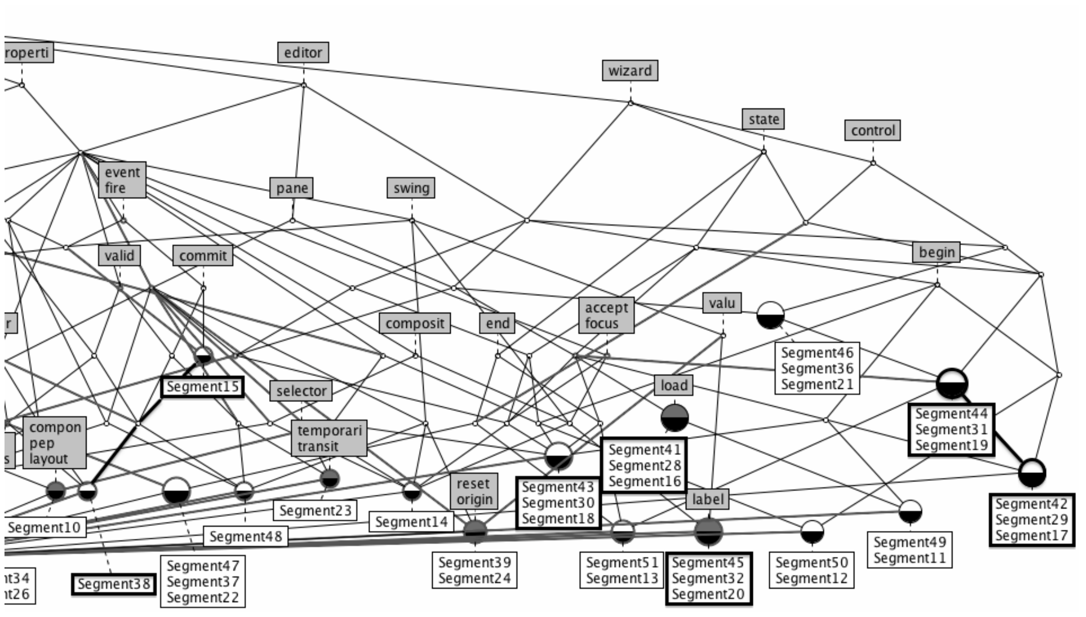


Figure 9. Excerpt of the Pooka FCA lattice for the scenario "New account new e-mail".

4.4.3. *RQ3: To what extent does SCAN correctly identify relations among segments?*

Figure 9 shows an excerpt of the Pooka FCA lattice for the scenario "New account new e-mail" and Table XII shows some of the relations identified by SCAN for this scenario. For example, SCAN identifies that Segments 16, 28, and 41 form a phase, *i.e.*, Phase 16 in Table XII, as they all activate the same feature, which is loading the state through the wizard editor pane. SCAN automatically labels this phase as "load state wizard editor pane".

Hence, Segment 16 activates Phase 16. The next segment in the trace, Segment 17, activates Phase 17. In the same way Segments 18, 19, and 20 activate respectively Phases 18, 19, and 20. Thus, the sequence of Segments 16 to 20 activate the sequence of Phases 16 to 20. However, the same sequence of phases is also activated with the sequence of Segments 28 to 32 as well as with the sequence of Segments 41 to 45. Thus, the three sequences of Segments 16 to 20, Segments 28 to 32, and Segments 41 to 45, activate the same features, *i.e.*, activate the features of Phases 16 to 20. Thus, SCAN identifies a macro-phase from the repeated execution of Phase 16 $\rightarrow$ Phase 17 $\rightarrow$ Phase 18 $\rightarrow$ Phase 19 $\rightarrow$ Phase 20.

Considering the automatic labels produced by SCAN, we observe that Phase 17 and Phase 19 activate features pertaining to the state of the wizard editor pane. However, Phase 17 is more specific as it is concerned with beginning states. SCAN identifies a sub-feature relation between Phase 17 and Phase 19, as shown in Figure 9. SCAN also reports that Segment 38 has a sub-feature relation with Segment 15. The former raises different property events, including property committing events, in common with Segment 15.

Table XIII reports the number of relations identified by SCAN in the six programs (we do not report numbers for Neuroph, as no relation was identified among its segments), with and without the numbers of sub/super relations. SCAN identifies 100 relations: 59 sub/super, 7 macro-phase, and 34 same feature relations. An agreement between SCAN and the participants occurs when the same relation is identified by SCAN and at least one of the participants. We do not show results when both participants agree, as the number of cases in which participants disagree is low (six and eight relations in case of no distinction and distinction, respectively).

We can conclude that, depending on whether we distinguish sub/super relations or not, the overall accuracy of SCAN in identifying relations between segments is 91% and 63%, respectively. When evaluating relations with distinction, the precision of SCAN is greater than 75% in the majority of the programs. The two exceptions are ArgoUML and Mars, and for both of these programs, the

Table XIII. Evaluation of the automatic relations.

Programs	Relations identified by SCAN	Sub/Super feature relations	Agreements with participant(s) without distinction btw. sub/super relations	Agreements with participant(s) with distinction btw. sub/super relations
ArgoUML	6	6	100%	33%
JHotDraw	9	5	100%	100%
Mars	22	18	100%	9%
Maze	12	1	100%	83%
Pooka	51	29	82%	78%
Total	100	59	91%	63%

proportion of detected sub-feature relations is extremely high with respect to other relation, *i.e.*, 100% and 82% for ArgoUML and Mars, respectively.

For ArgoUML, the majority of the detected relations involve class `MetaTypesMDRImpl`, which retrieves objects that represent the different UML types. SCAN labels Segment 2 as "*mdr meta impl types*", which is the general feature implemented by the class. SCAN labels the rest of the segments by specifying additional terms, *e.g.*, "*composite state meta impl synch mdr types*" for Segment 11. Thus, Segment 2 is identified as the super-feature of five other segments, as SCAN considers them as addressing more specific features. Both participants labeling ArgoUML traces produced similar label for Segment 2, *e.g.*, "*implementation dependent UML class type*" but also for the rest of the segments and, thus, considered all segments to implement the same feature. We observe a similar phenomenon for Mars, as shown in Table XIII, *i.e.*, when no distinction is made participants agree 100% with the identified relations.

We thus answer **RQ3**: To what extent does SCAN correctly identify relations among segments? as follows: SCAN identifies relations among segments with an overall precision of 63% and a precision greater than 75% in the majority of the programs.

5. APPLYING SCAN TO SUPPORT FEATURE LOCATION

According to Dit *et al.* [40] feature location is "*one of the most frequent maintenance activities undertaken by developers because it is a part of the incremental change process [41]*". Given the importance of feature location in the context of software maintenance tasks, we further explore how trace segmentation and labeling performed by SCAN can be used to support feature location to help developers in their everyday activity.

Consider a feature location techniques that uses dynamic analysis, as the Single Trace and Information Retrieval approach (SITIR) proposed by Liu *et al.* [42], which combines dynamic analysis and textual analysis based on Latent Semantic Indexing (LSI). Given a change request—*e.g.*, bug description—and an execution trace, the approach proposed by Liu *et al.* [42] ranks methods of the source code that appear in the execution trace based on their textual similarity with the change request—*e.g.*, the bug description or title. It is important to point out that feature location techniques aim at finding a starting point of the modification, *i.e.*, the "seed"—a method in the source code that is relevant for the change request and where developers will start the necessary modifications to implement the change request. The motivation for that is because, once the seed is known, the developer can identify the other methods that would be impacted by any change related to such a feature, *e.g.*, impacted by a bug fixing activity.

For this reason, the effectiveness of a feature location technique is evaluated in terms of the number of methods in the ranked list produced by the technique that a developer has to scrutinize before reaching any method belonging to the impact set of the feature. Such a method would be the seed for a modification. In order to perform this kind of evaluation, we require the availability of a gold set, *i.e.*, the set of all methods (and those methods only) that a developer should modify in order to fix a bug. The lower the number of methods to explore before finding the seed, the better the technique.

In this context, we are interested to evaluate whether SCAN can be used to reduce the burden of developers when identifying the set of methods impacted by a feature, once a feature location technique identifies one of these methods (*i.e.*, the seed). The conjecture is that methods related to a feature should be contained in one or few segments. Hence, to analyze the feature impact set, a developer could only focus on one or few segments instead of looking at the entire execution trace. In addition to that, we also want to investigate whether, instead of relying on feature location techniques, SCAN can be used as a standalone technique to automatically identify segments relevant for a query.

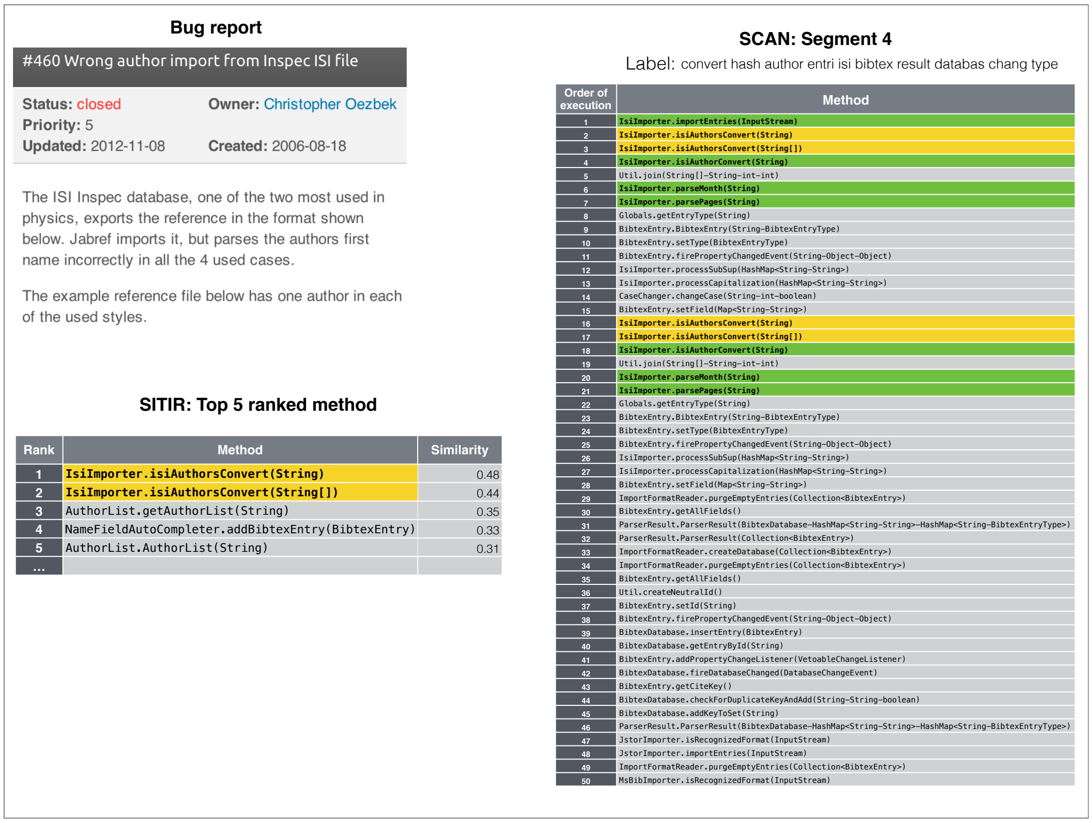
In the rest of this section we describe a typical scenario showing how SCAN can be used to support feature location (Section 5.1). Then, we evaluate the usefulness of SCAN through an empirical study, whose definition and planning are provided in Section 5.2, and whose results are reported and discussed in Section 5.3.

5.1. Typical scenario

Figure 10 shows an example of a bug report for JabRef and the top 5 ranked methods produced by SITIR. In a typical scenario the developer assigned to implement the changes will start by looking into the first method of the ranked list—*i.e.*, method `isiAuthorsConvert(String)` defined in `IsiImporter`—by trying to understand the source code and–or execution trace. The execution trace in this particular case consists of $13,616$ method calls.

To ease the analysis of the execution trace, a developer can use SCAN to segment it. Figure 10 shows one segment of this trace—the segment containing the top 1 ranked method provided by the feature location technique. The segment shows the methods in their order of execution, thus method `isiAuthorsConvert(String)` occurs two times (in positions 2 and 16) as it was executed twice. Since SCAN's segmentation is guided by the textual cohesion between methods, segment 4 can be regarded as the smallest, highly cohesive part of the trace activating the problematic feature that provides the context in which the top 1 ranked method is executed. In other words, rather than considering the entire execution trace, one may limit the context of a method to the segment in which it appears. From the methods of segment 4 one can quickly grasp that the author conversion is indeed performed in the context of importing a bibliographical entry. A further analysis of the methods contained in segment 4 reveals that the segment also contains a couple of other gold set methods—appearing at positions 1, 3, 4, 6, 7, 17, 18, 20, and 21. Indeed the developer fixing the wrong author import also fixed other related problems in the import of an ISI entry—the parsing of month and pages in particular. Thus, we conjecture that methods relevant to a change request are grouped in few segments. If this is indeed the case SCAN can be used to reduce the analysis of the trace to the analysis of few segments, *i.e.*, reduce the number of methods to be analyzed. This assumes that a feature location technique is used to guide the search and is able to retrieve the segments containing the gold set methods.

However, when no feature location technique is available to guide the search, SCAN can also be used to retrieve the segments that contain relevant methods by using the FCA lattice produced in the earlier stage and the title of the issue report as a query to guide the search. Figure 11 shows the partial FCA lattice corresponding to the trace of the example shown in Figure 10. We include the query as part of the set of objects and use the terms of the query to retrieve the segments it is related to. In other words, in order to identify the relevant segments, we look for the segments that share terms with the query. Thus, we start from the node representing the query and following the paths towards the top node the first two reachable nodes connect Query with Segment 3 (as they share the words *"isi"* and *"inspec"*) and Query with Segment 4 (as they share the words *"isi"* and *"author"*). Those are the segments containing the methods from the gold set. In general, the closer we are to the top node, the more segments we collect as we are less restrictive on the terms that those segments must contain. The closer we are to the Query node, the less segments we collect as we impose greater number of terms to be shared between them. Our conjecture is that SCAN can be used to automatically identify the segments containing the relevant methods. We also hypothesize that the automatically retrieved segments reduce the number of methods to be analyzed, compared to the analysis of the entire execution trace.


Bug report

#460 Wrong author import from Inspec ISI file

Status: closed **Owner:** Christopher Oezbek
Priority: 5
Updated: 2012-11-08 **Created:** 2006-08-18

The ISI Inspec database, one of the two most used in physics, exports the reference in the format shown below. Jabref imports it, but parses the authors first name incorrectly in all the 4 used cases.

The example reference file below has one author in each of the used styles.

SITIR: Top 5 ranked method

Rank	Method	Similarity
1	IsiImporter.isiAuthorsConvert(String)	0.48
2	IsiImporter.isiAuthorsConvert(String[])	0.44
3	AuthorList.getAuthorList(String)	0.35
4	NameFieldAutoCompleter.addBibtexEntry(BibtexEntry)	0.33
5	AuthorList.AuthorList(String)	0.31
...		

SCAN: Segment 4

Label: convert hash author entri isi bibtex result databas chang type

Order of execution	Method
1	**IsiImporter.importEntries(InputStream)**
2	**IsiImporter.isiAuthorsConvert(String)**
3	**IsiImporter.isiAuthorsConvert(String[])**
4	**IsiImporter.isiAuthorConvert(String)**
5	Util.join(String[]-String-int-int)
6	**IsiImporter.parseMonth(String)**
7	**IsiImporter.parsePages(String)**
8	Globals.getEntryType(String)
9	BibtexEntry.BibtexEntry(String-BibtexEntryType)
10	BibtexEntry.setType(BibtexEntryType)
11	BibtexEntry.firePropertyChangedEvent(String-Object-Object)
12	IsiImporter.processSubSup(HashMap<String-String>)
13	IsiImporter.processCapitalization(HashMap<String-String>)
14	CaseChanger.changeCase(String-int-boolean)
15	BibtexEntry.setField(Map<String-String>)
16	**IsiImporter.isiAuthorsConvert(String)**
17	**IsiImporter.isiAuthorsConvert(String[])**
18	**IsiImporter.isiAuthorConvert(String)**
19	Util.join(String[]-String-int-int)
20	**IsiImporter.parseMonth(String)**
21	**IsiImporter.parsePages(String)**
22	Globals.getEntryType(String)
23	BibtexEntry.BibtexEntry(String-BibtexEntryType)
24	BibtexEntry.setType(BibtexEntryType)
25	BibtexEntry.firePropertyChangedEvent(String-Object-Object)
26	IsiImporter.processSubSup(HashMap<String-String>)
27	IsiImporter.processCapitalization(HashMap<String-String>)
28	BibtexEntry.setField(Map<String-String>)
29	ImportFormatReader.purgeEmptyEntries(Collection<BibtexEntry>)
30	BibtexEntry.getAllFields()
31	ParserResult.ParserResult(BibtexDatabase-HashMap<String-String>-HashMap<String-BibtexEntryType>)
32	ParserResult.ParserResult(Collection<BibtexEntry>)
33	ImportFormatReader.createDatabase(Collection<BibtexEntry>)
34	ImportFormatReader.purgeEmptyEntries(Collection<BibtexEntry>)
35	BibtexEntry.getAllFields()
36	Util.createNeutralId()
37	BibtexEntry.setId(String)
38	BibtexEntry.firePropertyChangedEvent(String-Object-Object)
39	BibtexDatabase.insertEntry(BibtexEntry)
40	BibtexDatabase.getEntryById(String)
41	BibtexEntry.addPropertyChangeListener(VetoableChangeListener)
42	BibtexDatabase.fireDatabaseChanged(DatabaseChangeEvent)
43	BibtexEntry.getCiteKey()
44	BibtexDatabase.checkForDuplicateKeyAndAdd(String-String-boolean)
45	BibtexDatabase.addKeyToSet(String)
46	ParserResult.ParserResult(BibtexDatabase-HashMap<String-String>-HashMap<String-BibtexEntryType>)
47	JstorImporter.isRecognizedFormat(InputStream)
48	JstorImporter.importEntries(InputStream)
49	ImportFormatReader.purgeEmptyEntries(Collection<BibtexEntry>)
50	MsBibImporter.isRecognizedFormat(InputStream)

Figure 10. Bug#460 in JabRef: Wrong author import from Inspec ISI file.

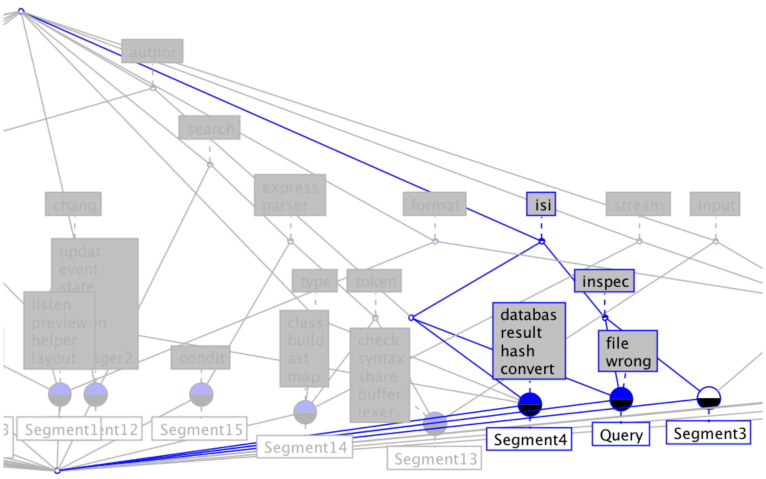


Figure 11. Bug#460 in JabRef: Resulting FCA lattice.

Table XIV. Program characteristics.

Program	Release Range	Issues	Traces with two or more gold set methods	Gold set methods
JabRef	2.0–2.6	39	17	280
muCommander	0.8.0–0.8.5	92	48	717
Overall		131	65	997

Table XV. Traces and Segments Characteristics.

	Number of method calls					
	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Traces	3K	57K	95K	95K	126K	264K
Segments	2	2	2	104	3	5K

5.2. *Empirical Study Definition and Planning*

The *goal* of the study is to assess the usefulness of SCAN for developers with the *purpose* of showing its usefulness when performing feature location tasks as a complement to a feature location technique or as a standalone technique. The *quality focus* is the possible effort reduction achieved when using SCAN, due to the smaller number of methods a developer should scrutinize. The *perspective* is of researchers interested in providing support to program comprehension by labeling and relating segments in execution traces.

5.2.1. *Study Set-Up*

This section details the study set-up, specifically describing the datasets that we use, *i.e.*, the execution traces and gold set methods of selected issue reports for two Java programs.

The objects of our evaluation are execution traces collected from two Java programs belonging to different domains. JabRef[10] is an open source bibliography reference manager. It uses the BibTeX file format and provides a user-interface to manage BibTeX files. muCommander[11] is a lightweight, cross-platform file manager with a dual-pane interface. It allows users to perform file operations on a variety of local and networked file systems, including FTP, Windows shares, and so on.

Table XIV summarizes characteristics of the programs, *i.e.*, the interval considered (*i.e.*, from release x to release y) the numbers of bugs occurred in such a time interval, the number of traces that include two or more gold set methods, and the total number of gold set methods. The execution traces were generated for the latter release—*i.e.*, 2.6 for JabRef and 0.8.5 for muCommander.

The choice of JabRef and muCommander for this study is related to the availability of execution traces, issue reports, and the associated gold set methods. The dataset was made publicly available by Dit *et al.* [43].

Overall, we analyzed a total of 65 execution traces from the two programs—17 from JabRef and 48 from muCommander. Table XV reports descriptive statistics about the numbers of method calls in the execution traces as well as in the segments that SCAN identifies.

5.2.2. *Experimental Design and Analysis*

The study aims at answering the following research questions:

- **RQ4:** *Does SCAN has a potential to support feature location?* This research questions aims at evaluating SCAN's ability to group gold set methods into a low number of segments thus reducing the number of methods to be analyzed by developers when limiting the analysis to

[10] http://jabref.sourceforge.net/
[11] http://www.mucommander.com/

the set of segments containing the gold set methods rather than the entire execution trace. The conjecture—as explained in Section 5.1—is that such segments would contain most of the methods related to the feature, hence the developer could easily determine the set of methods impacted when performing the change—*e.g.*, the bug fixing—by looking at the segments containing the seeds only.

- **RQ5:** *To what extent does SCAN support feature location tasks if used as a standalone technique?* This research question investigates whether it is possible to automatically retrieve segments containing relevant methods and evaluates the number of methods to be analyzed compared to the number of methods in the entire execution trace. We calculate the recall with respect to the segments containing the gold set methods as well as the recall with respect to the gold set methods.

To address **RQ4**, we first investigate whether methods from the gold set are grouped within few segments. To this end, we provide the number of segments containing the gold set methods and the total number of segments of the traces. The lower the number of segments containing the gold set methods the more grouped they are and thus the more potentially useful SCAN is. For example, the execution trace for the example bug report in Figure 10 is segmented by SCAN into 26 segments, and the gold set methods are concentrated in 2 of those segments—Segment 3 and Segment 4. Clearly, the sizes of the segments also impact the extent to which SCAN would help to reduce the effort: analyzing many small segments would not be effort-prone as analyzing few bigger segments (representing the execution of a whole feature), although the absence of cohesive segments would provide no guidance to the developer for knowing when to stop analyzing methods. To mitigate this problem, we consider the size of the segments in terms of total number of method calls and in terms of unique methods.

We estimate the number of methods in the segments containing the gold set methods and divide it by the number of methods in the entire execution trace. This ratio represents the number of methods to be analyzed if one is able to retrieve the segments containing the gold set methods. The lower such ratio, the better. In the above example, the total number of method calls is 52—2 in Segment 3 and 50 in Segment 4. The ratio is thus 0.0038 $(= 52/13,616)$.
We also estimate the ratio of the unique number of methods in the segments containing the gold set methods over the total number of methods in the execution trace. The lower the ratio the greater the gain in terms of number of unique methods that developers need to understand. This ratio indicates the upper bound limit for the reduced number of methods that an automatic technique retrieving the segments must return to developers. It is an upper bound as reducing more, *i.e.*, not analyzing some of those segments, would result in incomplete change implementation. Any technique that identifies additional segments would be decreasing the reduction of methods to be analyzed. Reaching the upper bound assumes that we have a perfect way to identify those and only those segments. For the above example, the unique number of methods called in Segments 3 and 4 is 34 and the unique number methods in the entire execution trace is 479 resulting in a ratio of 0.07 $(= 34/479)$. This ratio is a proxy for the effort that developers would need to spent if they concentrate on the segments containing the gold set methods rather than the entire trace.

To address **RQ5**, we use the labels of segments, the relations among the segments produced by SCAN, and the title of the bug report as search query to retrieve the segments having one or more terms in common with this query. For each trace, SCAN builds the FCA lattice using the segments of the trace while adding the title of the bug report (query) to the set of objects of the lattice. SCAN considers as objects segments and the query; attributes are the terms in the segments/query. By analyzing the resulting FCA lattice, SCAN identifies the segments in relation with the query: First, starting from the node representing the query and following the paths towards the top node SCAN collects all encountered nodes. Next, for all the collected nodes SCAN identifies the segments they are connected to. The set of collected segments contains all segments with which the query has same feature, sub-feature, or super-feature relation.

We evaluate the ability of SCAN to retrieve relevant segments using the gold set methods and calculating two types of recall. We calculate the recall with respect to the segments containing the

Table XVI. Number of gold set methods.

Program	Min	1st Qu.	Median	Mean	3rd Qu.	Max
JabRef	2	3	4	5.5	7	16
muCommander	2	2	3	6	7	35
Overall	2	2	3	5.9	7	35

gold set methods as well as the recall with respect to the gold set methods only. To calculate the recall for segments, we divide the number retrieved segments containing gold set methods by the total number of segments containing the gold set methods (see Equation 4). For the example shown in Figure 11, the recall for segments is 1 (*i.e.*, 100%) as both Segments 3 and 4 are retrieved.
To calculate the recall for methods, we divide the number retrieved gold set methods by the total number of gold set methods (see Equation 5). The recall for methods in the above example is also 1 $(= 7/7)$ as SCAN retrieves all the gold set methods.

$$Recall_{\text{Segments}} = \frac{\text{retrieved segments containing gold set methods}}{\text{total number of segments containing the gold set methods}} \quad (4)$$

$$Recall_{\text{Methods}} = \frac{\text{retrieved gold set methods}}{\text{total number of gold set methods}} \quad (5)$$

Here also we also provide a proxy for the effort that developers would need to spent if they concentrate on the segments retrieved by SCAN rather than the entire trace. This estimate of effort is expressed as the ratio of the number of methods to be analyzed if analyzing the methods contained in the retrieved by SCAN segments and the entire trace. For the above example, SCAN retrieves 5 segments with a total of 133 unique methods being called. The trace consists of 479 unique methods being called thus resulting in a ratio of 0.27 $(= 133/479)$, *i.e.*, 27%.

Finally, we also analyze how the recall varies when the number of terms in labels varies from 10 to 100. Previously, in Section 4, we limited the number of words in a label to ten as we were seeking to provide a concise summary of a segment—this constraint for conciseness was imposed by the purpose of the label, *i.e.*, help developers to quickly grasp the concepts of a segment. For the purpose of automatic feature location, we increase the number of words as the labels will be used to automatically retrieve the relevant segments and thus a higher amount of terms is not an issue.

5.3. *Study Results*

This section reports the results of our experimental study to address the research questions formulated in Section 5.2.2.

5.3.1. *RQ4: Does SCAN has a potential to support feature location?*

To answer this research question, we investigate whether multiple methods from the gold set are grouped within the same segments. Thus, from the 131 traces of JabRef and muCommander, we are interested in those containing at least two gold set methods, *i.e.*, 65 of those traces, see Tables XIV. Table XVI shows statistics of the numbers of gold set methods for those traces. We observe that the gold sets consist of around six methods on average, as shown by the column Mean of Table XVI.

Table XVII provides statistics on the distribution of the gold set methods across the different segments of the traces. We observe that the gold set methods are usually concentrated in only two segments, as shown by the column Mean of Table XVII (first row) while on average the total number of segments for a trace is close to 36 (second row). Therefore, we conclude that, indeed, methods of the gold set are grouped into segments and thus SCAN has the potential to guide developers to other methods useful to their feature location task.

Table XVII. Distribution of the gold set methods across the segments.

	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Number of segments containing the gold set methods	1	1	2	2.2	3	5
Overall number of segments in a trace	14	30	38	35.9	41	68
Percentage of the size of the segments containing gold set methods (over the size of the trace)	0%	1.56%	2.16%	2.48%	3.37%	6.47%
Unique number of methods appearing in segments required to understand (compared to the unique number of methods required to understand the entire trace)	0%	29.63%	44.77%	47.09%	61.92%	81.83%

As explained in Section 5.2.2, we also provide a rough estimate of the effort developers would save if they focus their understanding activity on segments containing the gold set methods rather than understanding the entire trace. The effort to understand a segment (respectively, a complete trace) is estimated by the number of unique method calls in that segment (respectively, trace). Table XVII presents statistics regarding the percentage of the sizes of the segments containing the gold set methods with respect to the overall sizes of the traces. The percentage is also provided in terms of unique methods. We conclude that the size of the segments containing the gold set methods is smaller than 3% of the size of the entire traces. If focusing the understanding on relevant segments only (*i.e.*, those containing methods from the gold set) rather than on the entire trace, we can reduce the number of methods to analyze by about 47%.

Consider again the example shown in Figure 10. Rather than analyzing the entire execution trace, the developers may focus on Segment 4, which provides the context in which method `isiAuthorsConvert(String)` is called. By looking at the method calls in Segment 4, they can understand that the author conversion is performed in the context of importing a bibliographical entry. They can also realize that, in general, importing an ISI entry also requires parsing the month and pages, which were not performed adequately. Hence, Segment 4 also contains other gold set methods, appearing at positions 1, 3, 4, 6, 7, 17, 18, 20, and 21, which were modified to fix problems in the import of an ISI entry, the parsing of month and pages in particular, while fixing the author conversion.

We thus answer **RQ4**: Does SCAN has a potential to support feature location? as follows: SCAN has the potential to be useful during feature location because it groups gold set methods in only two segments in general. Assuming that the segments containing the gold set methods can be retrieved, understanding those relevant segments saves about 53% of the methods that developers would need to understand compared to the entire execution traces.

5.3.2. RQ5: To what extent does SCAN support feature location tasks if used as a standalone technique?

Table XVIII shows the results of $Recall_{\text{Segments}}$, *i.e.*, the recall with respect to the segments containing the gold set methods, for different sizes of the labels (we vary the size from 10 to 100, step by 10). We observe that, for example, when the maximum number of terms in a label is 100, we can retrieve 75% of the segments containing the gold set methods; for 68% of the traces, we retrieve all segments—*i.e.*, 100% recall. The minimum retrieved segments is 0% because gold set methods are sometimes filtered in the preprocessing of the segmentation.

Table XIX shows $Recall_{\text{Methods}}$, *i.e.*, the recall with respect to the gold set methods (rather than the segments that contain them). Thus, considering again the case where the number of terms in a label is limited to 100, we observe that, on average, we retrieve 70% of the gold set methods. Table XX shows that analyzing the retrieved segments represents on average 57% of the methods that one would have to understand to analyze the entire trace.

Table XVIII. $Recall_{\text{Segments}}$: Retrieving segments containing gold set methods.

Label Size	Min	1st Qu.	Median	Mean	3rd Qu.	Max
10	0%	0%	0%	35.48%	100%	100%
20	0%	0%	33.33%	45.22%	100%	100%
30	0%	0%	50%	50.86%	100%	100%
40	0%	0%	50%	56.55%	100%	100%
50	0%	28.57%	100%	64.59%	100%	100%
60	0%	33.33%	100%	67.41%	100%	100%
70	0%	50%	100%	73.18%	100%	100%
80	0%	50%	100%	74.31%	100%	100%
90	0%	50%	100%	74.31%	100%	100%
100	0%	50%	100%	75.08%	100%	100%

Table XIX. $Recall_{\text{Methods}}$: Retrieving gold set methods.

Label Size	Min	1st Qu.	Median	Mean	3rd Qu.	Max
10	0%	0%	0%	36.05%	100%	100%
20	0%	0%	33.33%	43.26%	100%	100%
30	0%	0%	33.33%	46.72%	100%	100%
40	0%	0%	50%	53.27%	100%	100%
50	0%	20%	66.67%	58.88%	100%	100%
60	0%	33.33%	80%	62.36%	100%	100%
70	0%	33.33%	100%	68.18%	100%	100%
80	0%	33.33%	100%	69.24%	100%	100%
90	0%	33.33%	100%	69.24%	100%	100%
100	0%	33.33%	100%	70.01%	100%	100%

Table XX. Number of methods needed to understand the retrieved segments compared to the number of methods needed to understand the entire trace.

Label Size	Min	1st Qu.	Median	Mean	3rd Qu.	Max
10	0%	0%	3.88%	27.05%	50.24%	81.83%
20	0%	0.32%	30.11%	33.39%	67.07%	81.83%
30	0%	1.65%	38.08%	37.2%	67.07%	81.83%
40	0%	22.57%	43.73%	42.66%	68.61%	81.83%
50	0%	25.5%	54.93%	47.78%	71.82%	81.83%
60	0%	31.99%	57.32%	50.6%	77.3%	82.85%
70	0%	35.74%	63.16%	53.75%	77.3%	82.85%
80	0%	40.79%	63.16%	54.4%	77.3%	82.85%
90	0%	44.78%	65.08%	55.76%	77.3%	82.85%
100	0%	49.31%	65.08%	56.56%	77.08%	82.85%

Finally, we observe that increasing the size of the labels leads to more gold set methods and more segments that contain them to be retried. However, it also increases the number of methods to be analyzed. From Tables XVIII, XIX and XX, we conclude that a value ranging between 70 and 100 terms in the labels seems to be optimal as it retrieves close to 74% of the segments containing the gold set methods, which corresponds close to 69% of the gold set methods, while saving near 45% of the methods to analyze.

We thus answer **RQ5**: To what extent does SCAN support feature location tasks if used as a standalone technique? as follows: When no technique is used to guide developers, SCAN can retrieve relevant segments. For the analyzed traces, the recall with respect to the gold set methods is close to 69% while saving near 45% of the methods to analyze compared to the entire execution traces.

6. THREATS TO VALIDITY

This section discusses the threats to the validity of our evaluation and explains how we tried to mitigate them when possible.

Construct validity Construct validity concerns the relation between theory and observations. Our theory is that participants consistently understand execution traces and, thus, that an automated approach can accurately segment, label, and relate execution traces. Previous work already used trace segmentation for feature location, *e.g.*, [44]. Moreover, participants can perform maintenance tasks so we believe that our theory holds in general. In particular, threats to the construct validity of our evaluation could mainly be due to our evaluation of the capability of SCAN to label segments and to identify relations between them. For the former, we compare automatically generated labels with manually generated labels in **RQ2**. To limit bias due to subjectiveness, we also report results obtained by applying union or intersection over labels produced by multiple participants. The participants of the experiment are not the original developers of the studied programs. To address this threat to validity, we asked more than one participant to manually label the same segment. Note also that developers of large software programs may not be familiar with the entire program and thus would have been participant to the same threat.

In **RQ1**, we show that using an approach based on *tf-idf* to identify terms for labeling segments makes sense, as labels produced by participants when using reduced segment subsets with the most frequently invoked methods are not significantly different from those obtained when having the full segments available. As for the relations in **RQ3**, we asked participants to validate them and, because they do not know how our approach works, their bias is limited.

In **RQ4** and **RQ5**, we estimate a proxy of the effort a developer has to spend when performing feature location in terms of the number of methods to be analyzed. We are aware that this is a roughly estimate, because the actual effort could involve many factors, such as the length and complexity of those methods, the overall code complexity, quality of the lexicon, experience of the developer, etc. However, in a context in which the impact set will be determined by performing a basic understanding of each candidate method—*e.g.*, by looking at its signature and comments if any—such an approximation may result reasonable.

In **RQ4** and **RQ5**, we estimate a proxy of the effort that developers need to provide to fix a bug. This proxy measurement is based on the number of methods that developers need to analyze. In other words, we did not perform an experiment with actual developers to measure the time and effort required to fix the bug as such an experiment would require that the developers in the control group use the entire execution trace, which generally is overly large.

Internal Validity The internal validity of an evaluation is the extent to which a treatment changes the dependent variable. The internal validity of our empirical evaluation could be threatened by our choice of the traces to segment and label, and whose segments to relate as well as related thresholds (*e.g.*, the threshold used to merge two segments). We mitigated this threat by using different traces obtained from executing different scenarios on different programs. Also, participants confirmed the precision and recall of both the segment labels and their relations.

Last, but not least, we are aware that the performance of the trace segmentation approach can be affected by factors such as the choice of the LSI k value. In this context we have used the same value adopted in our previous work [19, 20] where the segmentation approach was introduced. In future work we plan to experiment the approach sensitivity to different k values.

External Validity The external validity of an evaluation relates to the extent to which we can generalize its results. Our empirical evaluation of the performances of SCAN is limited to six programs; we evaluate the usefulness of SCAN to support feature location on two programs. Yet, our approach is applicable to any other program. However, we cannot claim that the same results would be achieved with other programs. Different programs and different scenarios may lead to different results. Our choices reduce the threat to the external validity of our empirical evaluation. As explained in Section 4.2.2, participants involved in the evaluation of the performances of SCAN are not original developers of the analyzed programs, hence results might be different when considering people having a better knowledge of the systems.

Conclusion validity Conclusion validity threats deal with the relation between the treatment and the outcome. Wherever appropriate, we use statistical tests to support our claims. Specifically, for **RQ1**, we use permutation test, which is a non-parametric alternative to ANOVA, hence it does not require data to be normally distributed.

7. RELATED WORK

This section describes related work concerning (i) feature and concept location, and (ii) source code summarization.

7.1. Feature and Concept Location

Feature location aims at identifying subsets of a program source code activated when exercising a piece of functionality. Techniques using static analysis, dynamic analysis, and their combination, have been proposed in the literature. A static technique based on abstract dependencies graph was proposed by Chen and Rajlich [45]. Recently, Le *et al.* [46] proposed a multi-abstract concern localization where code units and textual descriptions are represented at multiple abstraction levels. Le *et al.* iteratively apply Latent Dirichlet Allocation (LDA) to create an abstraction hierarchy from the initial bags of words extracted from code units and textual descriptions. Results show that the multi-abstract VSM-based concern localization outperforms the original VSM-based concern localization. Also recently, Bassett *et al.* [47] focused on a static LDA-based feature location technique and propose a new approach to calculate term weighting. Rather than the common term weighting (*e.g.*, binary, term frequency, *tf-idf*), Bassett *et al.* propose to weight terms based on structural information extracted from the source code. Thus, terms derived from method names can be given more importance than terms derived from local variable names. Bassett *et al.* show that in the context of an LDA-based feature location technique the structural term weighting significantly improves the accuracy of the results. In our approach we use *tf-idf* but in future we plan to investigate structural term weighting.

Wilde and Scully [48] proposed a dynamic analysis technique for feature location. Eisenbarth *et al.* [44] combined the use of both static and dynamic data to identify features. We share with Eisenbarth *et al.* the use of both static and dynamic information. However, our purpose is different: while they aimed at performing feature location, our aim is to identify cohesive segments in execution traces, assign them a concept (by means labeling), and finally relate them to help developers to understand traces.

Antoniol and Guéhéneuc [49] presented a hybrid approach for feature location and reported its results on real-life, large, object-oriented, multi-threaded programs. They used knowledge filtering and probabilistic ranking to overcome the difficulties of getting rid of uninteresting events. The approach was improved [4] by using the notion of disease epidemiology. Later, Poshyvanyk *et al.*

[6] located features by combining information from the scenario-based probabilistic ranking with Latent Semantic Indexing (LSI).

Poshyvanyk *et al.* [7] used Formal Concept Analysis (FCA) and LSI to locate concepts in source code corresponding to a textual description, *e.g.*, description of a feature or a bug report. Given a query, *i.e.*, a summary of the description, source code elements are ranked using LSI. From the top-most elements in the ranked list, the top-most descriptive terms are selected and a concept lattice is built. As Poshyvanyk *et al.* did, our approach uses LSI and FCA. It also uses both static and dynamic information whereas the previous authors used static information only. When building FCA lattices, we operated at a higher level of abstraction, *i.e.*, in our case objects are segments of execution traces rather than methods. This is because our aim is different, *i.e.*, identify cohesive segments relevant for a feature rather than identify single methods related to a feature. Finally, we used relations among FCA concepts in the lattice to automatically identify relations among segments.

Rohatgi *et al.* [5] presented a hybrid approach for feature location. They used dynamic analysis to generate an execution trace by exercising a feature of interest. Then, they used static analysis to rank the methods of the dependency graph built from the methods invoked in the execution trace. They ranked the methods by relevance to the feature. The ranking mechanism guides developers to locate methods implementing a feature of interest, without the need for a deep understanding of the program.

Pirzadeh *et al.* [50] proposed a trace sampling framework. They used stratified sampling to obtain traces of reduced size wrt. the original trace by distributing the desired characteristics of an execution trace similarly in both the sampled and the original trace. They used random sampling techniques to generate sampled execution traces. However, random sampling may generate samples that are not representative of the original trace. They extended their approach [51] by extracting higher-level views that characterize the relevant information about execution traces.

Alawneh *et al.* [52] identify computational phases from inter-process communication traces of High Performance Computing (HPC) systems. The approach targets specifically HPC systems as the complexity of phase detection is higher due to multi-processing. The approach first identifies inter-process communication patterns and then groups the identified patterns into clusters, *i.e.*, phases. The phase detection is based on Shannon entropy and Jensen-Shannon divergence measure. The approach also identifies sub-phases using a threshold, the segmentation strength, to control the number of detected sub-phases. The higher the segmentation strength, the lower the number of sub-phases. The first aim of SCAN, *i.e.*, generating trace segments, is similar to what also Pirzadeh *et al.* [50, 51] and Alawneh *et al.* [52] did, although based on a different approach, incorporating conceptual similarity in a search-based optimization technique [19, 20]. In addition to that, as illustrated in this paper SCAN is also able to label segments and, using FCA, to determine relations between them.

Our work is also close to the work of Pirzadeh and Hamou-Lhadj [8] who divide execution traces into segments corresponding to the program's main execution phases (*e.g.*, initializing variables, performing a specific computation, etc.). The algorithm for phase detection is inspired by the psychology laws describing how the human brain groups similar items. Potential execution phases are identified by applying the similarity and continuity gravitational schemes. The similarity scheme reduces the distance between same method calls, where distance is defined using a mapping from the execution order of the method calls to an interval scale, *i.e.*, ruler distance. The continuity scheme reduces the distance between method calls in higher nested levels and the previous method calls. Applying the schemes may result in a rearrangement of the methods calls compared to the original trace, *i.e.*, the order of execution is not preserved. Pirzadeh and Hamou-Lhadj then use the K-means clustering algorithm to group potential phases thus identifying the execution phases. In contrast with their work, SCAN preserves the order of method calls when segmenting an execution trace and labels the resulting segments.

Asadi *et al.* [19] presented an approach based on genetic algorithms to split execution traces into cohesive segments. They identified concepts by finding cohesive and decoupled segments by applying LSI on source code. Although Asadi *et al.* [19] could identify concepts with high precision, the approach was computationally intensive and could not be applied on traces with thousands of method calls. Using dynamic programming, the approach was improved by Medini *et al.* [20] to improve its scalability. Our work builds upon this previous work on splitting execution traces into cohesive segments, in that it aims at labeling segments identified using the approach by Medini *et al.* [20] and at identifying relations among segments.

7.2. Source Code Summarization

Software artifact summarization consists of techniques extracting short descriptions from software artifacts to help developers during program comprehension. There are different approaches to summarize source code. Some of them use heuristics to extract structured or natural-language summaries [53, 54] whereas others use IR techniques to extract relevant keywords representing software artifacts [55, 56].

Sridhara *et al.* [53] proposed 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 [54] by using a classification of code into fragments, to generate a natural language description of "actions" related to each fragment. The authors identified three types of fragments: sequence fragments, conditional fragments and loop fragments.

Haiduc *et al.* [55, 56] applied and combined several automatic summarization techniques. In a reported case study, they found that a combination of techniques making use of the position of terms in software and traceability recovery techniques capture the meaning of methods and classes better than any other of the studied techniques. In addition, an experiment conducted with four developers revealed that the summaries produced using this combination are sensible.

De Lucia *et al.* [34] experimented the use of different IR techniques to extract keywords from source code artifacts. They compared the labeling obtained using simple Vector Space Models (VSM) and *tf* or *tf-idf* weighting schemes with those of more complex techniques, such as LSI. They used a manual labeling performed by 17 students as oracle against which to compare the various techniques. They found that simpler indexing techniques, such as VSM, outperform LSI, whose results were better only for larger artifacts in which LSI clustering capabilities help to reduce the noise.

Wang *et al.* proposed an approach that automatically segments source code methods into meaningful blocks for the purpose of automatic blank line insertion [57]. The approach uses the program structure and identifiers to identify consecutive statements that logically implement a high-level action. Examples of meaningful blocks are a sequence of statement that belong to the same syntactic category (*e.g.*, method call, variable declaration), a sequence of statements that are related through data flow, and a sequence of statements grouped in a while loop including the immediately preceding statements initializing variables that control the condition. Wang *et al.* also define a statement-pair similarity measure to segment syntactical blocks; the measure is based on the program identifiers—the use of words that constitute them and naming conventions—and has only three possible values. For each three consecutive statements there will be a segmentation if the similarity between the first two and the last two statements is different. Wang *et al.* study how developers insert blank lines to define heuristics that automatically mimics their behavior. In our work it is impossible to use similar approach as execution traces are not manually written. In contrast to the sliding window used by Wang *et al.* we start with one statement and we keep adding the following statements one by one as long as the fitness function is improved; we segment when adding a statement decreases the value of the fitness function.

We share with all software summarization techniques described above—and in particular keyword-based summarization techniques—the goal of representing artifacts with shorter descriptions. In particular, as De Lucia *et al.* [34], we use a simple VSM (properly complemented with some heuristics to remove noise) to label execution trace segments. The main difference between labeling source code artifacts and labeling execution traces is that execution traces must be (1) pruned, else utility methods and event handlers dominate the creation of labels, thus producing meaningless labels and (2) segmented, else labels would not be meaningful to developers. Finally, segments must be related with one another.

8. CONCLUSION

Program comprehension activities are crucial and preliminary to any maintenance or evolution tasks. Execution traces help developers to understand programs and relate methods (and methods calls) with user-visible features. However, execution traces are often overly large. Hence, they cannot be used directly by developers for program comprehension activities, in general, and feature location, in particular.

Consequently, we proposed SCAN, Segment Concept AssigNer, to support feature location by breaking execution traces into segments and labeling those segments to represent the features that they execute. SCAN also relates (1) segments activating the same feature, (2) segment(s) activating part of a feature activated by other segment(s), and (3) sequences of segments activating the same set of features. SCAN relies on dynamic programming to split traces into cohesive segments and to merge similar segments from multiple traces. Then, it labels the segments with relevant terms using Vector Space Model (VSM) and *tf-idf*. Finally, it uses Formal Concept Analysis (FCA) to identify relations among segments. SCAN thus provides developers with segmented traces, whose segments are labeled and related with one another.

To evaluate SCAN, we first conducted a study aimed at analyzing the ability of SCAN to accurately reduce the size of segments, identify labels, and identify relations between segments. After that, we conducted a second study aimed at investigating the usefulness of SCAN to support feature location tasks. In the first study, we asked 31 participants (professionals and students) to assign labels to segments extracted from six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka). First (**RQ1**), we investigated whether providing the participants with the most relevant methods only is sufficient to understand segments; we compare the quality of the labels and participant agreement with those obtained when full segments were used to produce labels. Then (**RQ2**), we compared manually-produced labels for 210 segments with those produced by SCAN. Finally (**RQ3**), we used SCAN to identify relations among segments and asked participants to validate them. Results of the empirical evaluation confirmed the ability of SCAN to select the most representative methods of a segment, thus reducing on average 92% of the information that participants must process while guaranteeing that close to 50% of the knowledge is preserved (**RQ1**)—the labels produced by participants when analyzing the reduced segments contained 50% of the information of the labels produced from the original segments. Results also showed that SCAN can automatically label segments with 69% precision and 63% recall when compared to the manual labels produced by the participants (**RQ2**). Finally, the results showed that participants agreed at 63% with the identified relations among segments identified by SCAN (**RQ3**).

In the second study we investigated how SCAN could help in a feature location task, when used in combination with a state-of-the-art feature location approach (**RQ4**), or when used as a standalone approach (**RQ5**). Our conjecture is that methods relevant for a feature would be grouped by SCAN in one or two segments, and therefore limiting the analysis to the methods in these segments would reduce the effort compared to analyzing the methods in the entire execution traces (**RQ4**). We also investigated whether SCAN is able to automatically retrieve these relevant segments and whether analyzing the retrieved segments reduces the number of methods to be analyzed compared to the entire execution trace (**RQ5**). Results show that in general relevant methods are grouped in two segments and analyzing only those segments reduces about 53% of the methods that developers would need to understand compared to the entire execution traces (**RQ4**). Results also show that, for the analyzed traces, SCAN can retrieve close to 69% of the relevant methods while reducing the number of methods to analyze by 43% compared to analyzing the entire traces (**RQ5**).

In summary, we showed that SCAN provides useful information to developers performing feature location tasks: it provides relevant segments, labels for these segments, and relations among segments. Future work includes investigating the applicability of SCAN to different object-oriented programming languages, for example C++. Also, we will investigate the applicability of SCAN as a re-documentation tool, and we plan to perform user studies in which developers use SCAN during their maintenance activities.

ACKNOWLEDGEMENT

The authors would like to express their gratitude to all the participants to the experiment who dedicated between one and two hours of their time to understand the purpose of execution trace segments, to label them, and to reason on the relations among segments.

REFERENCES

1. Kozaczynski V, Ning JQ, Engberts A. Program concept recognition and transformation. *IEEE Transactions on Software Engineering* 1992; **18**(12):1065–1075.
2. Biggerstaff T, Mitbander B, Webster D. The concept assignment problem in program understanding. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE CS Press, 1993; 482–498.
3. Koschke R, Quante J. On dynamic feature location. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2005; 86–95.
4. Antoniol G, Guéhéneuc YG. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering* 2006; **32**(9):627–641.
5. Rohatgi A, Hamou-Lhadj A, Rilling J. An approach for mapping features to code based on static and dynamic analysis. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2008; 236–241.
6. Poshyvanyk D, Guéhéneuc YG, Marcus A, Antoniol G, Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 2007; **33**(6):420–432.
7. Poshyvanyk D, Gethers M, Marcus A. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology* November 2012; **21**(4):23:1–23:34.
8. Pirzadeh H, Hamou-Lhadj A. A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2011; 221–230.
9. Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* September 2009; **35**(5):684–702.
10. Biermann A. On the inference of turing machines from sample computations. *Artificial Intelligence* 1972; **3**:181–198.
11. Reiss SP, Renieris M. Encoding program executions. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2001; 221–230.
12. Hamou-Lhadj A, Lethbridge T. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2006; 181–190.
13. Hamou-Lhadj A, Braun E, Amyot D, Lethbridge T. Recovering behavioral design models from execution traces. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2005; 112–121.
14. Safyallah H, Sartipi K. Dynamic analysis of software systems using execution pattern mining. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2006; 84–88.
15. Lo D, Khoo SC, Liu C. Efficient mining of iterative patterns for software specification discovery. *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2007; 460–469.
16. Lo D, Maoz S. Scenario-based and value-based specification mining: better together. *Automated Software Engineering (ASE)* 2012; **19**(4):423–458.
17. Agrawal H, Demillo RA, Spafford EH. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience* June 1993; **23**(6):589–616.
18. Zhang X, Gupta R, Zhang Y. Precise dynamic slicing algorithms. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2003; 319–329.
19. Asadi F, Di Penta M, Antoniol G, Guéhéneuc YG. A heuristic-based approach to identify concepts in execution traces. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010; 31–40.
20. Medini S, Galinier P, Di Penta M, Guéhéneuc YG, Antoniol G. A fast algorithm to locate concepts in execution traces. *Proceedings of the International Symposium on Search-based Software Engineering (SSBSE)*, 2011; 252–266.
21. Eisenbarth T, Koschke R, Simon D. Feature-driven program understanding using concept analysis of execution traces. *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 2001; 300–309.
22. Bellman RE, Dreyfus SE. *Applied Dynamic Programming*, vol. 1. Princeton University Press, 1962.
23. Cormen TH, Leiserson CE, Rivest RL. *Introductions to Algorithms*. MIT Press, 1990.
24. Marcus A, Poshyvanyk D, Ferenc R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering* 2008; **34**(2):287–300.
25. Medini S, Antoniol G, Guéhéneuc YG, Di Penta M, Tonella P. SCAN: an approach to label and relate execution trace segments. *Proceedings of Working Conference on Reverse Engineering (WCRE)*, 2012; 135–144.
26. Ng JKY, Guéhéneuc YG, Antoniol G. Identification of behavioral and creational design motifs through dynamic analysis. *Journal of Software Maintenance and Evolution: Research and Practice* 2010; **22**(8):597–627.
27. Porter MF. An algorithm for suffix stripping. *Program* 1980; **14**(3):130–137.
28. Baeza-Yates R, Ribeiro-Neto B. *Modern Information Retrieval*. Addison-Wesley, 1999.
29. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science* 1990; **41**(6):391–407.
30. Nasir Ali, Guéhéneuc YG, Antoniol G. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Transactions on Software Engineering* October 2012; **39**(5):725–741.

31. Poshyvanyk D, Marcus A. The conceptual coupling metrics for object-oriented systems. *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2006; 469–478.
32. Ville BGR. *Formal Concept Analysis*. Mathematical Foundations: Springer, 1999.
33. Siff M, Reps T. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering* November-December 1999; **25**:749–768.
34. De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Using IR methods for labeling source code artifacts: Is it worthwhile? *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2012; 193–202.
35. Yevtushenko SA. System of data analysis "concept explorer" ((in russian)). *Proceedings of the National Conference on Artificial Intelligence KII-2000*, 2000; 127–134.
36. Zéphyrin Soh, Zohreh Sharafi, Bertrand van den Plas, Cepeda Porras G, Guéhéneuc YG, Antoniol G. Professional status and expertise for uml class diagram comprehension: An empirical study. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2012; 163–172.
37. Baker RD. Modern permutation test software. *Randomization Tests*, Edgington E (ed.), Marcel Decker, 1995.
38. Miller GA. WordNet: A lexical database for English. *Communications of the ACM* 1995; **38**(11):39–41.
39. Hintze JL, Nelson RD. Violin plots: A box plot-density trace synergism. *The American Statistician* 1998; **52**(2):181–184.
40. Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 2013; **25**(1):53–95.
41. Rajlich V, Gosavi P. Incremental change in object-oriented programming. *IEEE Software* 2004; **21**(4):62–69.
42. Dapeng L, Andrian M, Denys P, Vaclav R. Feature location via information retrieval based filtering of a single scenario execution trace. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2007; 234–243.
43. Dit B, Holtzhauer A, Poshyvanyk D, Kagdi H. A dataset from change history to support evaluation of software maintenance tasks. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013; 131–134.
44. Eisenbarth T, Koschke R, Simon D. Locating features in source code. *IEEE Transactions on Software Engineering* March 2003; **29**(3):210–224.
45. Chen K, Rajlich V. Case study of feature location using dependence graph. *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 2000; 241–250.
46. Le TDB, Wang S, Lo D. Multi-abstraction concern localization. *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2013; 364–367.
47. Bassett B, Kraft NA. Structural information based term weighting in text retrieval for feature location. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2013; 133–141.
48. Wilde N, Scully MC. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance* 1995; **7**(1):49–62.
49. Antoniol G, Guéhéneuc YG. Feature identification: A novel approach and a case study. *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2005; 357–366.
50. Pirzadeh H, Shanian S, Hamou-Lhadj A, Mehrabian A. The concept of stratified sampling of execution traces. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2011; 225–226.
51. Pirzadeh H, Hamou-Lhadj A, Shah M. Exploiting text mining techniques in the analysis of execution traces. *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2011; 223–232.
52. Alawneh L, Hamou-Lhadj A. Identifying computational phases from inter-process communication traces of hpc applications. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2012; 133–142.
53. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for Java methods. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2010; 43–52.
54. Sridhara G, Pollock L, Vijay-Shanker K. Automatically detecting and describing high level actions within methods. *Proceeding of the International Conference on Software Engineering (ICSE)*, 2011; 101–110.
55. Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010; 223–226.
56. Haiduc S, Aponte J, Moreno L, Marcus A. On the use of automated text summarization techniques for summarizing source code. *Proceedings of Working Conference on Reverse Engineering (WCRE)*, 2010; 35–44.
57. Wang X, Pollock L, Vijay-Shanker K. Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of Software: Evolution and Process* 2014; **26**(1):27–49.