A Heuristic-based Approach to Identify Concepts in Execution Traces

Fatemeh Asadi*, Massimiliano di Penta**, Giuliano Antoniol*, and Yann-Gaël Guéhéneuc***
fatemeh.asadi@polymtl.ca dipenta@unisannio.it antoniol@ieee.org yann-gael.gueheneuc@polymtl.ca

* SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada
** Department of Engineering, University of Sannio, Benevento, Italy
*** Ptidej Team – DGIGL, École Polytechnique de Montréal, Québec, Canada

***Abstract*—Concept or feature identification, *i.e.*, the identification of the source code fragments implementing a particular feature, is a crucial task during software understanding and maintenance. This paper proposes an approach to identify concepts in execution traces by finding cohesive and decoupled fragments of the traces. The approach relies on search-based optimization techniques, textual analysis of the system source code using latent semantic indexing, and trace compression techniques. It is evaluated to identify features from execution traces of two open source systems from different domains, JHotDraw and ArgoUML. Results show that the approach is always able to identify trace segments implementing concepts with a high precision and, for highly cohesive concepts, with a high overlap with the manually-built oracle.**

***Keywords*—Concept location, dynamic analysis, information retrieval.**

I. Introduction

Software systems often lack an adequate and up-to-date documentation. Therefore, developers must resort to reading the system source code, without specific tool support but code browsers, to understand the systems and perform their maintenance and evolution tasks. In some cases, code understanding is supported by static analysis and–or visualizations built upon static information. In other case, debugging can be used to understand the behavior of a system in a particular context and–or to locate a fault. However, manually browsing of source code, inspecting an execution trace or debugging long sequences of instructions are time consuming and daunting tasks.

Concept or feature location and identification aim at helping developers to perform their maintenance and evolution tasks, by identifying abstractions (*i.e.*, features) and the location of the implementation of these abstractions. Specifically, they aim at identifying *code fragments*, *i.e.*, set of method calls in traces and the related method declarations in the source code, responsible for the implementation of domain concepts and–or user-observable features [1], [2], [3], [4], [5]. The literature reports approaches built upon static [6] and dynamic [7], [5] analyses; Information Retrieval (IR) [4] and hybrid (static and dynamic) [8] techniques.

This paper proposes a novel approach to identify cohesive and decoupled fragments in execution traces, which likely participate in implementing concepts related to some features. A typical problem for which the proposed approach can be beneficial is the following. Suppose a failure has been observed when executing a particular scenario of a software system; unfortunately the likelihood to reproduce the execution conditions for that failure are very low. Maintainers are then faced with the problem of analyzing the execution trace produced by that scenario and identifying high level abstractions that likely participate in the feature producing the unwanted behavior.

To deal with the above described problem, the proposed approach identifies concepts composing an execution scenario by grouping together methods that are (i) sequentially invoked together/in sequence and (ii) cohesive and decoupled from a conceptual point of view. The underlying assumption is that, if a specific feature is being executed within a complex scenario (*e.g.*, "Open a Web page from a browser" or "Save an image in a paint application"), then the set of methods being invoked is likely to be conceptually cohesive, decoupled from those of other features, and sequentially invoked. We use conceptual cohesion and coupling from Marcus *et al.* [9] and Poshyvanyk *et al.* [10].

The approach works as follows. First, we index the source code of each method of a system textually. Then, we instrument and exercise the system to collect execution traces for some scenarios related to different features and, therefore, containing sets of different concepts. We compress the traces to remove utility and cross-cutting concepts and to abstract repetitions of the same sub-sequences of methods. Finally, we apply a search-based optimization technique, *i.e.*, a genetic algorithm, to split the compressed traces into cohesive and decoupled fragments. We ensure performances by parallelizing the algorithm over multiple computers.

Overall, the contributions of this paper are:

1) A novel approach combining IR techniques, dynamic analysis, and search-based optimization techniques to identify concepts into execution traces;
2) An empirical study that shows the applicability and the performances of the proposed approach in identifying concepts into execution traces of two systems, JHotDraw and ArgoUML. Results indicate that the

approach is able to identify concepts (with a precision in most cases greater than 80%), while the overlap with a manually-built oracle varies depending on the cohesiveness of the concepts to be identified.

The remainder of the paper is organized as follows. Section II presents related work. Section III describes the approach. Section IV presents an empirical study and Section V report its results and some discussions. Section VI concludes the paper and outlines future work.

II. Related Work

Although, many feature and concept identification approaches exist, none of these approaches attempts to identify concepts in a system trace *automatically*.

In their pioneering work, Wilde and Scully [7] presented the first approach to identify features by analyzing execution traces. They used two sets of test cases to build two execution traces, one where a feature is exercised and another where the feature is not. They compared the execution traces to identify the feature in the system. Similarly, Wong *et al.* [11] analyzed execution slices of test cases to identify features in source code. Wilde's original idea was later extended in several works [1], [4], [12], [13] to improve its accuracy by introducing new criteria on selecting execution scenarios and by analyzing the execution traces differently.

Chen and Rajlich [14] developed an approach to identify features using Abstract System Dependencies Graphs (ASDG). In C, an ASDG models functions and global variables as well as function calls and data flow in a system source code. Chen and Rajlich identified features using the ASDG following a precise manual process. In contrast to Wilde and Scully's work, Chen and Rajlich used only static data to identify features and a manual process.

Eisenbarth *et al.* [15] combined previous approaches by using both static and dynamic data to identify features. In a following work, Eisenbarth *et al.* [13] introduced an approach to feature identification using test cases.

Salah and Mancoridis [16] used both static and dynamic data to identify features in Java systems. They went beyond feature identification by creating feature-interaction views, which highlight dependencies among features. Their work was extended to allow feature identification and evolution analysis in large-scale systems, *e.g.*, Mozilla [17].

More recent pieces of work focused on a combination of static and dynamic data [8], [4], in which, essentially, the problem of features location from multiple execution traces is modeled as an IR problem, which has the advantage to simplify the location process and, often, improves accuracy [4]. Yet, Liu *et al.* [18] showed that a single trace suffices to build an IR system and locate useful data. Execution traces were also used to mine aspects by Tonella and Ceccato [5].

We share with previous work the use of dynamic data and IR techniques to identify features. However, instead of querying traces using an IR technique, *e.g.*, similar to Poshyvanyk *et al.* [4], we determine cohesive and decoupled fragments likely being relevant to a concept *automatically*. Our approach is based on two conjectures not yet fully investigated: (1) methods helping to implement a concept are likely to share some linguistic information; (2) methods responsible to implement a feature are likely to be called close each other in an execution trace. Therefore, the conceptual coupling of methods participating in a concept should be high and these methods should appear relatively close together in the execution trace. The first conjecture is grounded on the findings published in [4] and other publications based on IR to locate features and concepts. IR-inspired works assume some form of commonalities between a query and linguistic information of entities. Locality of concept manifestation in traces is more questionable, however we believe unlikely that a user-observable feature or concept, not constituting a crosscutting concern, will be uniformly spread in a trace.

III. The Approach

This section describes the proposed approach to identify concepts by analyzing execution traces. The approach consists in five steps. First, the system is instrumented. Second, the system is exercised to collect execution traces. Third, the collected traces are compressed to reduce the search space that must be explored to identify concepts. Fourth, each method of the system is represented by means of the text that it contains. Fifth, a search-based optimization technique is used to identify, within execution traces, sequences of method invocations that are related to a concept.

A. Steps 1 and 2 – System Instrumentation and Trace Collection

First, the software system is instrumented using the *instrumentor* of MoDeC. MoDeC is a tool to extract and model sequence diagrams from Java systems [19]. The MoDeC instrumentor is a dedicated Java bytecode modification tool implemented on top of the Apache BCEL bytecode transformation library[1]. It inserts appropriate and dedicated method invocations in the system to trace method/constructor entries/exits, taking care of exceptions and system exits (`System.exit(int)`). It also allows the user to add to the traces tags containing meta-information *e.g.*, delimiting and labeling sequences of method calls related to some specific features being exercised.

Following the user manual (or use-case documents, if available) of the system to be analyzed, an execution scenario is composed of a sequence of cohesive steps. For example, exercising a Web browser could consist in a sequence of the following steps (i) open the browser; (ii) insert a URL and access a Web page; and (iii) save the Web page into a local HTML file. We exercise the instrumented

[1]http://jakarta.apache.org/bcel/

system to collect execution traces by following execution scenarios. Resulting traces are text files listing method calls and including the class of the object caller, the unique ID of the caller, the class of the receiver, its unique ID, and the complete signature of the method.

B. Step 3 – Pruning and Compressing Traces

Usually, execution traces contain methods invoked in most scenarios, *e.g.*, methods related to logging. Even in a single execution trace of an application with graphical user interface, mouse tracking methods will largely exceed all other method invocations. It is likely that such methods are not related to any particular concept, *i.e.*, they are utility methods. These methods do not provide useful information for developers when locating a concept, because they are common in many concepts. They are similar to low-discriminating terms occurring in many documents when applying a IR technique. Such terms are penalized by indexing measures like *tf-idf* [20].

Similarly, we built the distributions of the frequencies of method occurrences to remove too-frequent methods. We then prune out the methods having a frequency greater than $Q3 + 2 \cdot IQR$, where $Q3$ is the third quartile (75% percentile) of the distribution and IQR is the inter-quartile range. An alternative approach to deal with these methods is aspect mining [5], [21], because such methods can constitute crosscutting concerns. We do not use aspect mining because we are interested in pruning these methods to identify concepts, not in crosscutting concerns.

Traces often contain repetitions of one or more method invocations, for example `m1(); m1(); m1();` or `m1(); m2(); m1(); m2();`. A repetition does not introduce a new concept, thus we compress traces using the Run Length Encoding (RLE) algorithm to remove repetitions and keep one occurrence of any repetition only. The two previous examples would become `m1()` and `m1(); m2()`, respectively. Compression is performed for any sub-sequences of method invocations having an arbitrary length. Other encoding schema such as suffix trees or LZH algorithm are likely to produce even better results in a future work.

C. Step 4 – Textual Analysis of Method Source Code

To determine the conceptual cohesion of methods, our approach uses the Conceptual Cohesion metric defined by Marcus *et al.* [9]. We extract a set of terms from each method by tokenizing the method source code and comments, pruning out special characters, programming language keywords, and terms belonging to a stop-word list for the English language. (We assume that comments appearing on top of the method declaration belong to the following method.)

We split compound terms following the Camel Case naming convention at each capitalized letter, *e.g.*, `getBook` is split into `get` and `book`. Then, we stem the obtained simple terms using a Porter stemmer [22].

Once terms belonging to each method have been extracted, we index these terms using the *tf-idf* indexing mechanisms [20]. We obtain a term–document matrix, where documents are all methods of all classes belonging to the system under study and where terms are all the terms extracted (and split) from the method source code.

Finally, we apply Latent Semantic Indexing (LSI) [23] to reduce the term–document matrix into a concept–document matrix. The meaning of "concept" in LSI is different from that of "concept" in concept location. In LSI, a concept is one of the orthonormal dimensions of the LSI space. Following Marcus *et al.* [9], we compute the conceptual cohesion of methods in a class in the LSI subspace to deal with synonymy, polysemy, and term dependency. The chosen size of the LSI subspace is 50.

D. Step 5 – Search-based Concept Location

We now segment execution traces into conceptually-cohesive segments related to the feature being exercised (and thus to a specific concept). Determining a (near) optimal splitting of a trace into segments is NP-hard. Therefore, we use a genetic algorithm to perform the splitting and parallelize its computations.

1) Choice of the Optimization Technique: We experimented different techniques: hill climbing, simulated annealing, and genetic algorithms (GAs). We chose to use GAs because they outperformed other techniques due to the characteristics of the search space.

A GA may be defined as an iterative procedure that searches for the best solution to a given problem among a constant-size population [24]. The search starts from an initial population of individuals, represented by finite strings of symbols (the *genome*), often randomly generated. At each evolution step, individuals are evaluated using a *fitness function* and selected using a *selection mechanism*. High-fitness individuals have the highest reproduction probability. The evolution (*i.e.*, the generation of a new population) is affected by two genetic operators: the *crossover operator* and the *mutation operator*. The crossover operator takes two individuals (the *parents*) of one generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*) in the new generation. The mutation operator prevents the convergence to local optima: it randomly modifies an individual's genome (*e.g.*, by flipping some of its symbols).

2) Use of the Optimization Technique: Our representation of an individual is a bit-string as long as the execution trace in which we want to identify some feature-related concepts. Each method invocation is represented as a "0", except the last method invocation in a segment, which is represented as a "1". For example, the bit-string

$$\underbrace{00010010001}_{11}$$

$$SegmentCohesion_k = \frac{\sum_{i=begin(k)}^{end(k)-1} \sum_{j=i+1}^{end(k)} similarity(method_i, method_j)}{(end(k) - begin(k) + 1) \cdot (end(k) - begin(k))/2} \quad (1)$$

$$SegmentCoupling_k = \frac{\sum_{i=begin(k)}^{end(k)} \sum_{j=1, j<begin(k) \text{ or } j>end(k)}^{l} similarity(method_i, method_j)}{(l - (end(k) - begin(k) + 1)) \cdot (end(k) - begin(k) + 1)} \quad (2)$$

$$fitness(individual) = \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{SegmentCohesion_k}{SegmentCoupling_k} \quad (3)$$

Table I
EXAMPLE OF GA INDIVIDUAL REPRESENTATION (SECOND COLUMN).

Method Invocations	Repr.	Segments
TextTool.deactivate()	0	
TextTool.endEdit()	0	
FloatingTextField.getText()	0	
TextFigure.setText-String()	0	1
TextFigure.willChange()	0	
TextFigure.invalidate()	0	
TextFigure.markDirty()	1	
TextFigure.changed()	0	
TextFigure.invalidate()	0	
TextFigure.updateLocation()	0	2
FloatingTextField.endOverlay()	0	
CreationTool.activate()	1	
JavaDrawApp.setSelectedToolButton()	0	
ToolButton.reset()	0	
ToolButton.select()	0	
ToolButton.mouseClickedMouseEvent()	0	3
ToolButton.updateGraphics()	0	
ToolButton.paintSelectedGraphics()	0	
TextFigure.drawGraphics()	0	
TextFigure.getAttributeString()	1	

means that the trace, containing 11 method invocations, is split into three segments decomposed into the first four method invocations, the next three, and the last four. An real example of segment splitting[2] is shown in Table I.

The mutation operator randomly chooses one bit in the representation and flips it over. Flipping a "0" into a "1" means splitting an existing segment into two segments, while flipping a "1" into a "0" means merging two consecutive segments. The crossover operator is the standard 2-points crossover. Given two individuals, two random positions x, y with $x < y$ are chosen in one individual's bit-string and the bits from x to y are swapped between the two individuals to create a new offspring. The selection operator is the roulette-wheel selection. We use a simple GA with no elitism, *i.e.*, it does not guarantee to retain best individuals across subsequent generations. We set the population size to 200 individuals and a number of generations of 2,000 for shorter traces (those of JHotDraw) and 3,000 for longer ones (those of ArgoUML). The crossover probability was set to 70% and the mutation to 5%, which are widely used values in many GA applications.

A fitness function drives the GA to produce individuals that represent (near) optimal splittings of a trace into segments likely to relate to some concepts. In our fitness function, we use the software design principles of cohesion and coupling, already adopted in the past to identify modules in software systems [25], although we use conceptual (*i.e.*, textual) cohesion and coupling measures [9], [10], rather than structural cohesion and coupling measures.

Segment cohesion is the average (textual) similarity between any pair of methods in a segment k and is computed using the formulas in Equation 1 where $begin(k)$ is the position (in the individual's bit-string) of the first method invocation of the k^{th} segment and $end(k)$ the position of the last method invocation in that segment. The similarity between two methods is computed using the cosine similarity measure over the LSI matrix extracted in the previous step. Thus, it is the average of the similarity defined by [9], [10] to all pairs of methods in a given segment.

Segment coupling is the average similarity between a segment and all other segments in the trace, computed using Equation 2, where l is the trace length. As our conjecture is that a concept should be implemented by method calls locally close each other, on the one hand the algorithm favors the merging of consecutive segments containing methods with high average conceptual similarity. On the other hand, the algorithm penalizes solutions where consecutive segments are highly coupled together. The segment coupling represents, for a given segment, the average similarity between methods in that segment and those in different ones.

Finally, for a trace split into n segments, the fitness function is shown in Equation 3.

3) Parallelization of the Optimization Technique: One of the main advantages of GAs with respect to other optimization techniques is the possibility of parallelizing their computations, *e.g.*, the evaluations of the fitness of different individuals. In our approach, we use parallelization to reduce computation time. (However, a detailed study of the performances is out of scope of this paper and will be treated in a future work).

In our experiments, we distributed computations over a network of five servers and nine workstations. Servers are connected in a Gigabit Ethernet LAN while workstations are connected to a LAN segment at 100 MBit/s and talk to servers at 100 Mbit/s. Servers and workstations run CentOS

[2]The segment splitting shown in Table I has been obtained randomly and does not correspond to an actual "good" solution.

Table II
STATISTICS FOR THE TWO SYSTEMS.

Systems	NOC	KLOC	Release Dates
ArgoUML v0.18.1	1,267	203	30/04/05
JHotDraw v5.4b2	413	45	1/02/04

Table III
STATISTICS FOR THE COLLECTED TRACES.

Systems	Scenarios	Original Size	Cleaned Sizes	Compressed Sizes
ArgoUML	Start, Create note, Stop	34,746	821	588
	Start, Create class, Create note, Stop	64,947	1066	764
JHotDraw	Start, Draw rectangle, Stop	6,668	447	240
	Start, Add text, Draw rectangle, Stop	13,841	753	361
	Start, Draw rectangle, Cut rectangle, Stop	11,215	1206	414
	Start, Spawn window, Draw circle, Stop	16,366	670	433

5, 64 bits; memory varies between four and 16 Gbytes. Workstations are based on Athlon X2 Dual Core Processor 4400; the five servers are either single or dual Opteron. The distributed architecture comprises one workstation taking charge of distributing the GA individuals to other computers by means of socket connections. On the slave computers, a server receives the computation requests and the individuals, computes the value of the fitness function and returns the value back to the central computer. Only the fitness of new individuals, with respect to previous generations, are computed. The fitness values of individuals already evaluated are not recomputed but retrieved from a hash table storing the previous values.

Distribution over several computers is crucial to ensure acceptable computation time. With the described configuration, a single run takes about one hour. Scalability on very large traces will require different computation architectures (*e.g.*, sharing information between slaves) and possibly dividing a large trace into chunks with approaches inspired by overlapping time windows as in digital signal processing. This possibility will be studied in a future work.

IV. Empirical Study Description

We report an empirical study evaluating the proposed concept location approach. The *goal* of this study is to analyze the novel concept location approach based on dynamic data, with the *purpose* of evaluating its capability of identifying meaningful concepts. The *quality focus* is the accuracy and completeness of the identified concepts. The *perspective* is that of researchers who want to evaluate how the proposed approach can be used during maintenance and evolution. The *context* consists of an implementation of our approach and of the execution traces extracted from two open source systems, JHotDraw and ArgoUML.

A. Context

The context of our study are execution traces from ArgoUML and JHotDraw. Figure IV highlights main characteristics of the two systems. *ArgoUML*[3] is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. The ArgoUML project started in September 2000 and is still active. We analyzed release 0.19.8. *JHotDraw*[4] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. We analyzed release 5.1.

We generate traces by exercising various scenarios in the two systems. Table IV-A summarizes the scenarios and shows that the generated traces include from 6,000 up to almost 65,000 method invocations. The compressed traces include from 240 up to more than 750 method invocations. By exercising these scenarios, we do not want to identify concepts related to the systems' startup, *Start*, and closing, *Stop*. Therefore, in the following, when naming the scenarios and their associated features, we no longer include the Start and Stop concepts.

The GA was implemented using the *Java GA Lib*[5] library.

B. Building the Oracle

We need an oracle to assess the accuracy and completeness of the identified concepts. We build such an oracle by manually tagging the execution traces. Two "Start" and "Stop" tags enclose the method invocations related to a particular concept. While executing the instrumented system, before and after a step in the execution scenario (*e.g.*, Draw rectangle), the user, through a command in the instrumentor interface, inserts the appropriate tags in the execution trace and then continues to exercise the instrumented system with the next steps of the scenario. Consequently, the collected traces are composed of a sequence of method invocations interleaved with tags separating the invocations belonging to different steps.

C. Research Questions

This study aims at answering the three following research questions:

- **RQ1:** *How stable is the GA, through multiple runs, when identifying concepts into execution traces?* Approaches based on GAs could suffer from the randomness of the search: the initial individuals are randomly generated and the crossover, mutation, and selection operators are influenced by random choices. However, it is desirable that the representation, operators, fitness,

[3]http://argouml.tigris.org
[4]http://www.jhotdraw.org
[5]http://sourceforge.net/projects/javagalib/

and other settings (*e.g.*, population size and stopping criteria) be chosen so that multiple runs of the GA yields to similar solutions.

- **RQ2:** *To what extent the identified concepts match the ones in the oracle?* We are interested to evaluate the extent to which the identified segments overlaps with the ones in our oracle, obtained by manually tagging the traces.
- **RQ3:** *How accurate is the identification of concepts in execution traces?* Finally, we are interested to evaluate the extent to which the identified segments are accurate, *i.e.*, how many of the method invocations that they contain are also in the oracle and how many are not.

D. Study Settings and Analysis Method

To answer **RQ1**, we evaluate the extent to which the segments identified in multiple runs of the GA, and occurring in the same position of the trace, overlap each other. Let us consider a compressed trace composed of N method invocations $T \equiv m_1, \ldots m_N$ and partitioned at run i of the GA in k_i segments $s_{1,i} \ldots s_{k,i}$. For each segment $s_{x,i}$ obtained at run i, and for all the segmentations obtained at run $j \neq i$, we compute the maximum overlap between $s_{x,i}$ and the segments obtained at run j as follows:

$$max(Jaccard(s_{x,i}, s_{y,j})), y = 1 \ldots k_j$$

where:

$$Jaccard(s_{x,i}, s_{y,j}) = \frac{|s_{x,i} \cap s_{y,j}|}{|s_{x,i} \cup s_{y,j}|}$$

and where union and intersection are computed considering method invocations occurring at a given position in the trace. Stability is evaluated by means of descriptive statistics computed across the above obtained overlap values.

RQ2 is answered similarly to **RQ1** but, in this question, we compare the overlap between manually-tagged segments in the execution traces with segments identified by our approach. Specifically, given the segments determined by the tags in the trace (our oracle) and given the segments obtained by an execution of the system, we compute the overlap between each manually-tagged segment in the trace and the most similar automatically-identified segment.

Finally, **RQ3** is addressed like **RQ2**, with the only difference that we use precision instead of the Jaccard score, because we are interested in evaluating the accuracy of our approach. Precision is defined as follows:

$$Precision(s_{x,i}, s_{y,o}) = \frac{|s_{x,i} \cap s_{y,o}|}{|s_{y,o}|}$$

where $s_{x,i}$ are segments obtained by our approach and $s_{y,o}$ are segments in the corresponding trace in the oracle.

V. Results and discussion

This section reports the results of our experimental evaluation: the collected data and their analyses to address the previous research questions.

A. RQ1: How Stable is the GA across Multiple Runs?

We assess the stability of the GA by computing the average similarity of the segments identified in ten different runs of the approach. Table IV shows the similarity results. Overall, the similarity averages for JHotDraw range between 55% and 95%, with median values ranging between 70% and 84% . They are slightly higher for ArgoUML, between 80% and 83%. Thus, we conclude that, despite the potentially large size of the search space, our approach is able to generate stable segments across multiple runs. In addition, increasing the number of generations and the population size would potentially further increase the approach stability.

B. RQ2: To What Extent the Identified Concepts Match the Ones in the Oracle?

To address **RQ2**, we evaluate the extent to which the segments actually reflect features as they were manually tagged when executing the instrumented system to generate the execution traces.

For some features, *e.g.*, drawing a rectangle or a circle, the average (and median) Jaccard overlap is very high, suggesting that the features are implemented through sequences of very cohesive methods. Yet, other features exhibit lower overlaps. These lower overlaps do not mean that our approach was unable to successfully identify the features. Indeed, in some cases, for example the scenarios *Add text* in JHotDraw and *Create note* in ArgoUML, the features are realized by adapting a textual-editing feature as a shape-drawing feature, using the Adapter design pattern. The feature adaptation produces sequences of methods with a low cohesion, which our algorithm tend to split. As a consequence, the resulting overlaps are appropriately low.

In other cases, in particular with traces from ArgoUML, a large trace segment corresponding to a feature is split into two or more segments by our approach. Thus, the overlap between the (larger) manually-tagged segment and the corresponding automatically-identified segment is low. A manual study of such cases revealed that the manually-tagged segment is indeed composed of several smaller and cohesive sub-concepts that our algorithm tend to split, as illustrated and discussed in the following subsection.

C. RQ3: How Accurate is the Identification of Concepts in Execution Traces?

The right side of Table V reports the precision of the identified segments with respect to the manually-tagged ones. Precision is often very high, with median values in most cases above 85% and very often equal to 100%.

Lower precision values sometimes occur with explainable reasons. For example, in the scenario (2) of JHotDraw, composed of *Add text* and *Draw rectangle*, the two features are implemented using a very similar sequence of method invocations, making them hard to distinguish. Because these features are executed one after the other, our search-based

Table IV
DESCRIPTIVE STATISTICS OF SIMILARITY AMONG SEGMENTS OBTAINED IN TEN DIFFERENT RUNS.

Systems	**Scenarios/Features**	**Similarity Averages**				
		Min.	**Max.**	**Mean**	**Median**	σ
ArgoUML	(1) Add note	0.69	0.95	0.84	0.83	0.07
	(2) Add class, Add note	0.65	0.98	0.80	0.80	0.06
JHotDraw	(1) Draw rectangle	0.55	0.96	0.76	0.76	0.12
	(2) Add text, Draw rectangle	0.54	0.93	0.72	0.70	0.10
	(3) Draw rectangle, Cut rectangle	0.73	0.93	0.85	0.84	0.05
	(4) Spawn window, Draw circle	0.67	0.86	0.76	0.76	0.04

Table V
SIMILARITY (JACCARD OVERLAP AND PRECISION) BETWEEN SEGMENTS IDENTIFIED BY THE APPROACH AND FEATURES TAGGED IN THE TRACE.

Systems	**Scenarios**	**Features**	**Jaccard**					**Precision**				
			Min.	**Max.**	**Mean**	**Median**	σ	**Min.**	**Max.**	**Mean**	**Median**	σ
ArgoUML	(1)	Add note	0.15	0.39	0.28	0.27	0.08	0.91	1.00	0.97	1.00	0.04
	(2)	Create class	0.11	0.28	0.22	0.25	0.05	1.00	1.00	1.00	1.00	0.00
	(2)	Create note	0.22	0.56	0.35	0.31	0.14	1.00	1.00	1.00	1.00	0.00
JHotDraw	(1)	Draw rectangle	0.63	0.93	0.84	0.89	0.13	0.89	1.00	0.96	1.00	0.06
	(2)	Add text	0.21	0.31	0.26	0.27	0.05	0.27	0.36	0.32	0.34	0.04
	(2)	Draw rectangle	0.53	0.70	0.63	0.61	0.06	0.61	1.00	0.69	0.66	0.13
	(3)	Draw rectangle	0.42	0.76	0.64	0.72	0.14	0.73	1.00	0.94	1.00	0.11
	(3)	Cut rectangle	0.16	0.23	0.22	0.23	0.02	1.00	1.00	1.00	1.00	0.00
	(4)	Draw circle	0.54	0.96	0.85	0.88	0.14	0.81	1.00	0.91	0.95	0.09
	(4)	Spawn window	0.07	0.41	0.20	0.16	0.11	1.00	1.00	1.00	1.00	0.00

optimization technique is unable to split the trace into segment similar to the ones from the oracle. Consequently, the precision of *Add text* drops to a median value of 34% and that of *Draw rectangle*, usually very high in other scenarios, is only 66%.

D. Discussion

We analyze in detail some results to understand how the approach split the traces into segments. We focus on cases where the Jaccard score is low. In other cases, we know that the segments are meaningful because they are consistent with the oracle. Table VI shows excerpts of three segments. (Due to lack of space, we cannot report complete segments.)

The *Add class* feature of ArgoUML was matched with a very low Jaccard score. The manual tags in the trace delimited a sequence of 199 method invocations. The approach split this sequence into 5 segments comprising in a total of 172 method invocations, out of which 16 invocations occurred before the tag and, thus, do not belong to the oracle. The remaining $199 - 172 + 16 = 43$ invocations were grouped in small segments mainly related to GUI-event handling. In details, the five segments are related to (1) creation of the objects responsible for handling the class diagram through an instance of the Factory design pattern; (2) adding the class to the project; (3) adding the class to the current name-space; (4) setting properties of the class through a Façade design pattern; and, (5) handling the persistence of the diagram in the XMI file representing the UML diagram.

For the *Create note* feature of ArgoUML, the tagged segment is composed of 88 method invocations while the best matching segment identified by our approach is composed of 50 methods. The identified segment deals with the creation of a note, *i.e.*, creation of the object through a Factory, addition to the project, setting of its property. When compared to the *Add class* feature, only one segment was identified instead of five because the segment for creating a note is shorter than that of adding a class (50 invocations vs. 172) and because this smaller number of method invocations has a higher cohesion than that of the *Add class* feature. In addition, 32 of the remaining $88 - 50 = 38$ methods belong to the end of the trace and were not put in the same segment, while the sequence of these a methods continued after the tag with 24 other invocations. The continuation of the sequence *after* the tags means that the oracle is not precise enough. We explain this lack of precision by the extensive use of multi-threading in ArgoUML.

All methods related to setting properties through the Façade design pattern were not put in a same segment by our approach because these methods were invoked in a loop, in which each iteration of the loop contained a slightly different sequence of invocations. Consequently, (1) the RLE compression algorithm was not able to group together the various loop iterations and (2) the various iterations were not cohesive and thus the trace was split in several segments. We will explore in future work more complex compression techniques to deal with such cases.

The *Cut rectangle* feature of JHotDraw has been tagged as a sequence of 172 method invocations. However, in the best case shown in Table V, only 39 of these methods were grouped together by our approach, *i.e.*, the methods belonging to the last part of the tagged segment. We inspected this sequence and discovered that it is related to (1) add the rectangle content to the clipboard, (2) modify

Table VI
EXCERPT OF SEGMENTS IDENTIFIED BY THE APPROACH.

Create note (ArgoUML)	**Spawn window** (JHotDraw)	**Cut rectangle** (JHotDraw)
FacadeMDRImpl.isSingleton(...)	JavaDrawApp.createTools(...)	StorableOutput.close()
FacadeMDRImpl.isUtility(...)	MySelectionTool.MySelectionTool(...)	Clipboard.Clipboard()
CoreFactory.getCoreFactory()	TextFigure.TextFigure()	Clipboard.getClipboard()
CoreFactoryMDRImpl.buildComment(...)	TextFigure.setAttribute(...)	Clipboard.setContents(...)
CoreFactoryMDRImpl.createComment()	FigureAttributes.FigureAttributes()	CutCommand.deleteSelection()
CoreFactoryMDRImpl.initialize(...)	FigureAttributes.set(...)	BouncingDrawing.removeAll(...)
ModelEventPumpMDRImpl.flushModelEvents()	TextFigure.changed()	BouncingDrawing.figureRequestRemove(...)
UndoCoreHelperDecorator.addAnnotatedElement(...)	TextFigure.invalidate()	AnimationDecorator.removeFromContainer(...)
ModelEventPumpMDRImpl.flushModelEvents()	TextFigure.updateLocation()	AnimationDecorator.invalidate()
ClassDiagramGraphModel.addNode(...)	TextTool.TextTool(...)	AnimationDecorator.removeFigureChangeListener(...)
ClassDiagramGraphModel.canAddNode(...)	TextTool.TextTool(...)	AnimationDecorator.changed()
FacadeMDRImpl.isAInterface(...)	TextFigure.TextFigure()	AnimationDecorator.invalidate()
FacadeMDRImpl.isASubsystem(...)	TextFigure.setAttribute(...)	AnimationDecorator.release()
Project.getRoot()	FigureAttributes.FigureAttributes()	RectangleFigure.removeFromContainer(...)
ModelManagementFactory.getModelManagementFactory()	FigureAttributes.set(...)	RectangleFigure.removeFigureChangeListener(...)
ModelManagementFactoryMDRImpl.getRootModel()	TextFigure.changed()	RectangleFigure.changed()
CoreHelperMDRImpl.isValidNamespace(...)	TextFigure.invalidate()	RectangleFigure.release()
FacadeMDRImpl.getModel(...)	TextFigure.updateLocation()	RectangleFigure.removeFromContainer(...)
FacadeMDRImpl.isAModel(...)	ConnectedTextTool.ConnectedTextTool(...)	RectangleFigure.removeFigureChangeListener(...)
FacadeMDRImpl.isAFeature(...)	ConnectedTextTool.ConnectedTextTool(...)	RectangleFigure.changed()
...	...	...

the properties of the drawn rectangle so that it appears as "cut" in the painter, and (3) update the menu commands (*e.g.*, the command "Paste" is now enabled). The preceding sequence of $172 - 39 = 133$ methods was split in many small segments in which GUI events and actions performed by clicking the mouse button are interleaved, resulting in a sequence of loosely cohesive invocations.

The *Spawn window* feature of JHotDraw includes, in the manually-tagged segment, 197 method invocations; the segment with the highest overlap only included, however, 72 of these invocations. This sequence of 72 method invocations is actually related to re-sizing and re-adjusting figures in the panel while spawning the window. The remaining invocations (at the end of the trace) keep out by our approach are mainly related to restoring and setting-up the status of the menu commands.

Finally, as previously explained for the *Add text* feature of JHotDraw, the low Jaccard score and low precision are due to the high similarity between the sequences of methods of the *Add text* and *Draw rectangle* features, which leads our approach to put together both features in a segment of 168 method invocations.

On the one hand, the previous discussion highlights the capability of our approach to split execution traces into conceptually cohesive segments, despite the low Jaccard overlap with respect to manually-tagged segments. On the other hand, it shows some difficulties in identifying concepts in execution traces, due to:

- design patterns and, in general, object-orientation mechanisms (*e.g.*, polymorphism, dynamic binding), which make traces for different features almost identical (*e.g.*, *Add text* and *Draw rectangle* in JHotDraw);
- imprecision when generating and tagging traces due to multi-threading;
- the compression algorithm that is unable to group loop iterations consisting of slightly different sequences of method invocations.

In particular, despite the good results obtained by our approach when analyzing traces from JHotDraw (both with the Jaccard score and in precision), the extensive use of inheritance and design patterns in JHotDraw explain the lower results when compared to those obtained with ArgoUML. Inheritance and design patterns lead to the generation of many method invocations not directly related to a feature, but *supporting* and–or *enabling* the implementation of this feature. Consequently, these method invocations can appear in many different segments related to different features and thus can be a confounding factor for our approach.

Another difficulty of trace-based concept location approaches is to deal with method invocations related to GUI and system events. For example, hundreds of method invocations in both ArgoUML and JHotDraw execution traces correspond to GUI events, such as `mousePressed(...)`. These methods are not feature-specific and can appear almost anywhere in a trace and could lead to different segmentation across different runs. We deal with these methods by compressing the traces, removing sub-sequences of such methods, and using conceptual cohesion and coupling measures [9], [10], which lead to the creation of small segments containing only such method invocations.

E. Threats to validity

We now discuss the threats to validity that can have affected our empirical study.

Construct validity threats concern the relation between theory and observation. In this study, they are mainly due to measurement errors. The traces are automatically produced by executing the instrumented systems against some scenarios. Thus, the information contained in the traces is reliable. However, multi-threading could change the ordering of method calls in different traces exercising the same sub-scenarios. The performances of the proposed approach are evaluated by using the Jaccard overlap, already used in the past to evaluate concept location approaches [17] and by using the standard IR precision measure, because we are also interested to split the trace into segments that only contain methods related to the feature of interest.

We only performed a preliminary assessment of the meaning of the identified concepts, by manually analyzing sequences of method invocations belonging to different segments. In future work, we plan to use automated techniques to label segments and thus better help the maintainer by assigning meanings to segments automatically.

Threats to *internal validity* concern confounding factors that could affect our results. The manually-tagged traces that we use as oracle pose such a threat. Indeed, it is possible that tags would appear in slightly different positions in the traces obtained by exercising the same scenarios in different runs. The slight different positions result from multi-threading, as well as from method invocations related to mouse and other GUI events. In particular, extra method calls related to GUI events or other uncontrollable system events could be interleaved in the traces.

Methods declared in class libraries could also introduce "noise" in our approach. For example, calls to methods from the Java class libraries frequently occur in the traces obtained in our experiments. They do not occur frequently enough to be discarded as "utility" method calls yet are not related to interesting concepts. Therefore, in future work, we will consider adding these methods in our list of stop-words.

A last threat to internal validity relates to the intrinsic randomness of GAs. However, in **RQ1**, we showed that, overall, results are quite stable across different GA runs.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study.

Threats to *external validity* concern the possibility to generalize our results. We studied two systems having different size and belonging to different domains. However, we are aware that this is a first study aimed at validating the proposed approach and that we only split traces on a small sample of scenarios for the two software systems. Other traces could possibly lead to different results. Also, further validation on a larger set of different systems is desirable. Yet, within its limits, our results confirm the stability and precision of our approach for concept location.

A final remark concerns the complexity of our approach and computation times. Although this is a proof of concept, we are aware that excessive computation times or complexity may prevent further studies and practical application. On average, identifying concepts in a compressed trace of about 400 methods on a single high end PC (*i.e.*, with at least 4GB RAM) took about one day; when GA was mapped on multiple serves as described, the time went down to 20 minutes. Clearly, to make the approach appealing, we need to improve scalability both in time and space as well as in the possibility to handle longer traces.

VI. Conclusions and Future Work

This paper presented an approach to locate automatically concepts in execution traces by splitting traces into cohesive segments representing concepts related to a software system features. The approach relies on definitions of conceptual cohesion and coupling from the literature [9], [10] and on a search-based optimization technique, based on a genetic algorithm, to find (near) optimal splittings of traces into segments.

The approach has been applied and evaluated on two open source systems, ArgoUML and JHotDraw. Results showed that the approach is stable, and, overall, locates concepts with a high precision. Precision tend to drop for features realized using very similar sequences of methods, as sometimes happens in JHotDraw, where different kinds of shapes are drawn essentially in a same way. The overlaps between a manually-built oracle and the automatically-located segments vary depending on the cohesion of the features being analyzed, as the approach tends to split traces related to large features into smaller segments related to cohesive sub-concepts.

Future work will follow different directions. First, we are improving the proposed approach to increase its performance by better tuning the search-based optimization and the text indexing techniques. Also, we want to assign automatically meaningful labels to trace segments identified by the approach to help maintainers understand their meanings. Finally, we will carry other empirical studies to evaluate the approach on traces obtained from different systems.

VII. Acknowledgements

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution and in Software Patterns and Patterns of Software) and by G. Antoniol's Individual Discovery Grant.

References

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

[2] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "The concept assignment problem in program understanding," in *Proceedings of the* 15^{th} *International Conference on Software Engineering*, IEEE Computer Society Press / ACM Press, May 1993, pp. 482–498.

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

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

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

[6] N. Anquetil and T. Lethbridge, "Extracting concepts from file names: A new file clustering criterion," in *Proceedings of the* 20^{th} *International Conference on Software Engineering*, IEEE Computer Society Press, May 1998, pp. 84–93.

[7] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," in *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, January-February 1995, pp. 49–62.

[8] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proceedings of the* 16^{th} *International Conference on Program Comprehension (ICPC)*, IEEE Computer Society Press, June 2008.

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

[10] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*. Philadelphia, Pennsylvania, USA: IEEE CS Press, 2006, pp. 469 – 478.

[11] W. E. Wong, S. S. Gokhale, and J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software*, vol. 54, no. 2, pp. 87 – 98, 2000, special Issue on Software Maintenance.

[12] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," Software Engineering Research Center, Tech. Rep., 2004.

[13] A. D. Eisenberg and K. D. Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *proceedings of the* $21^{s}t$ *International Conference on Software Maintenance*. IEEE Press, September 2005, pp. 337–346.

[14] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of the* 8^{th} *International Workshop on Program Comprehension*, IEEE Computer Society Press, June 2000, pp. 241–252.

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

[16] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: From object-interactions to feature-interactions," in *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, IEEE Computer Society Press, September 2004, pp. 72–81.

[17] M. Salah, S. Mancordis, G. Antoniol, and M. Di Penta, "Towards employing use-cases and dynamic analysis to comprehend Mozilla," in *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Press, September 2005, pp. 639–642.

[18] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York NY USA: ACM, 2007, pp. 234–243.

[19] J. Ka-Yee Ng, Y.-G. Guéhéneuc, and G. Antoniol, "Identification of behavioral and creational design motifs through dynamic analysis," *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, December 2009.

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

[21] M. Marin, A. van Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, 2007.

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

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

[24] D. E. Goldberg, *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

[25] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch Tool." *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 193–208, 2006.