Scenariographer: A Tool for Reverse Engineering Class Usage Scenarios from Method Invocation Sequences

Maher Salah, Trip Denton, Spiros Mancoridis, Ali Shokoufandeh, Filippos I. Vokolos
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{msalah, tdenton, spiros, ashokouf, filip}@cs.drexel.edu

Abstract

Typical documentation for object-oriented programs includes descriptions of the parameters and return types of each method in a class, but little or no information on valid method invocation sequences. Knowing the sequence with which methods of a class can be invoked is useful information especially for software engineers (e.g., developers, testers) who are actively involved in the maintenance of large software systems.

This paper describes a new approach and a tool for generating class usage scenarios (i.e., how a class is used by other classes) from method invocations, which are collected during the execution of the software. Our approach is algorithmic and employs the notion of canonical sets to categorize method sequences into groups of similar sequences, where each group represents a usage scenario for a given class.

1 Introduction and motivation

Software undergoes continuous modification in response to changes in requirements, repairs of faults, performance enhancements, and so on. To be effective, software engineers must understand how the software works before they attempt to modify it. Unfortunately, the size and complexity of the software complicate the job of the software engineer, who may have to expend a significant amount of effort to comprehend the intricacies of the source code.

To help software engineers become more effective in understanding large software systems, the software engineering research community has responded with the development of techniques and tools such as source code analysis, software clustering, dynamic analysis of software execution, profiling, program slicing, and visualization. To see, from a high level, how existing tools and techniques can help a software engineer understand a software system better, consider the following scenario. Assume that a software engineer, who is unfamiliar with the source code, is asked to repair a fault in the 'Save' feature of a word processing application. The engineer can analyze the source code, perform software clustering, and visualize the results to get an overview of the software's design or architecture. Separately, the engineer can instrument the source code and execute the 'Save' feature to see which parts of the code implement this feature. Having obtained an overview of the software's structure and having narrowed the scope of his search to the most closely related code, the engineer can then proceed with the modification of the code to repair the fault.

A problem that software engineers often encounter, for which an effective, practical solution does not exist yet, is trying to understand how a class is used through its class interface. For example, assume that the software component under scrutiny is a class called `File`. This class has an interface that consists of the methods `open()`, `read()`, `write()`, `close()`, and the constructor method `File()`. In trying to understand how the `File` class works, one of the questions that needs to be answered is: what is a valid sequence of method invocations? Is the sequence <`open()`, `read()`, `close()`> a valid sequence, or should a `write()` be performed following a `read()`?

Often, there are method invocation sequences that represent various *usage scenarios* for a given class. A usage scenario describes how a class is used, through its interface, by other classes. For example, for the `File` class we have the following method invocation sequences and corresponding usage scenarios:

- The sequence < `open`, `close` >, which can be used to check the existence of a file.

- The sequence < `open`, `read`+, `close` >, which can be used to read the contents of a file. (+ indicates one or more repetitions.)
- The sequence < `open`, `write`+, `close` >, which can be used to write the contents of a file.
- The sequence < `open`, `read`+, `write`+, `close` >, which can be used to read a file, modify its content and, store the modified content back into the file.

Of course, these are just some of the many usage scenarios for the `File` class.

Software engineers who need to understand how to use the various classes of the software, would benefit from the availability of information that describes, in an easy to understand way, the valid method invocation sequences for a given class of the system.

The problem of knowing the valid method invocation sequences would be alleviated if every software interface had a specification, or at the very least a set of examples, that codified how its component could be used. Many programs have documentation (*e.g.*, Java programs have `javadoc` specifications) that describes the parameters and return types of each method in a class or the entry points of an API for a subsystem. Unfortunately, this type of documentation does not describe valid method invocation sequences.

In this paper, we describe a new approach for identifying a small subset of method invocation sequences that represent usage scenarios for a class. We have developed a tool based on this approach, called *Scenariographer*, which we have used to analyze some widely-used classes with encouraging results.

The intuition for our approach is based on the observation that the *method invocation sequences of a particular class usage scenario share certain similarities, but are fairly dissimilar from the sequences that correspond to other usage scenarios.* For example, considering the method invocation sequences and usage scenarios for the `File` class that were described above, we observe that all of the sequences used for just reading the contents of a file are of the form < `open`, `read`+, `close` >, while the sequences used for reading, manipulating, and storing the contents of a file are of the form < `open`, `read`+, `write`+, `close` >.

Our approach is algorithmic and relies on runtime information to identify the methods invoked for profiled instances of a class. The algorithm computes the distance between every pair of method invocation sequences using a well-defined distance measure (*i.e.*, edit distance of ordered sequences). Using this distance measure, the algorithm then computes the *canonical set* of method sequences, which is a small subset of the sequences that *best characterizes the elements of the original set*. By definition, each element of the canonical set is associated with a group of method sequences, called the *canonical group*, representing a possible usage scenario for a given class. The theoretical framework of our technique is based on a non-linear optimization formulation of canonical sets [10, 11]. An approximate solution to the optimization can be computed efficiently using semidefinite programming techniques [14] and rounding [27].

Our approach does not guarantee that for any set of method sequences it will correctly identify all class usage scenarios. However, as the results of a case study show in Section 5, the approach has been fairly effective in practice. We have been able to analyze software applications consisting of thousands of objects and methods to obtain usage scenarios for some widely-used classes.

The remaining of the paper is organized as follows: Section 2 reviews related work. Section 3 describes the design and implementation of *Scenariographer*. The high-level architecture of the tool is presented, followed by a description of the subsystems that comprise *Scenariographer*. Section 4 describes the algorithms for computing canonical sets and classifying the method invocation sequences into groups that represent the class usage scenarios. Section 5 presents the results of a case study, which includes two Java applications. Finally, Section 6 summarizes the work presented in this paper and discusses our plans for future work.

2 Background

Our focus on class usage scenarios has been motivated by the popularity and success of use cases in software engineering practices. Software engineers find use cases to be a convenient way to describe how a system is used by external users [12]. Operating at a different abstraction level, class usage scenarios aim to describe how a class is used by other classes of the system.

Although there is a substantial body of work on use cases, we are not aware of any work that specifically addresses the problem of analyzing object-oriented software systems to determine usage scenarios for a given class, either through static or dynamic analysis.

The automata theory and artificial intelligence communities have studied the problem of inferring finite state machines from examples (*e.g.*, strings). Some people refer to this problem as 'grammar generation', while others refer to it as 'inductive inference'. Recently, as a result of the increased popularity of finite state machines in connection with various object-oriented design methods (*e.g.*, UML [12]) this problem has attracted the interest of the software engineering community. Researchers have extended some of the basic approaches for generating finite state machines and applied them to domain-specific prob-

lems. The work by Ammons *et al.*, which we describe at the end of this section, is the one that is closest related to our work. What follows in this section overviews representative work in this area, which we believe provides necessary context for our approach.

A general paradigm of inductive inference was established by Gold, including the commonly-used criterion for a successful inference called 'identification in the limit' [16]. Work done within the general paradigm established by Gold was surveyed and explained by Angluin and Smith [5].

Biermann and Feldman [7] developed a method that, given a finite set of input-output pairs that sufficiently characterizes some finite-state computable function, finds the machine that realizes that function. The method includes an adjustable parameter k that allows one to vary the precision and complexity of the synthesized machine. At low settings of k, the machine tends to have few states, some non-determinism, and may yield a number of possible outputs for any given input. At higher values of k, the synthesized machine has more states, but is deterministic and has a precise representation of the desired input-output pairs.

An application of inductive inference is the auto-programming system developed by Biermann and Krishnaswamy [8]. This system constructs programs from example computations supplied by the user. The sample computations are in the form of condition-instruction pairs, with the conditions being Boolean expressions on the variables of the program. Programs can be represented as directed graphs with instructions as nodes and transitions as edges. The essential problem addressed by Biermann and Krishnaswamy is how to label instructions that appear multiple times in the sample computations, so that the end result is something more interesting than a linear directed graph.

The auto-programming system was the basis for an algorithm developed by Koskimies and Makinen to synthesize state machines from event trace diagrams [17]. A UML event trace diagram describes the order in which certain events are sent from one object to another. Motivated by the fact that event trace diagrams are not readily available for many legacy software systems, Systa and Koskimies proposed the use of runtime information as a way to generate event trace diagrams [26]. These trace diagrams can then be used with the Koskimies and Makinen algorithm to synthesize a state machine, described above, thus creating a state machine that describes the behavior of the various classes of objects of a legacy system. Note that the work by Systa and Koskimies solves a different problem than the one we have solved. Our emphasis is on characterizing class usage scenarios from method invocation sequences.

Ammons *et al.* developed a machine learning approach, called *specification mining*, for discovering formal specifications of the protocols that code must obey when interacting with an application program interface (API) or abstract data type (ADT) [3]. Specification mining, which also uses runtime information, summarizes the frequent interaction patterns as state machines. To overcome some analysis obstacles, it requires human expertise for defining flow dependencies, a non-trivial activity. Further, to deal with computational issues associated with the generation of state machines for large software system, it operates on small interaction scenarios, and not complete execution traces.

3 Scenariographer

This section describes the design and implementation of *Scenariographer*, the tool that collects runtime information and produces a small set of method invocation sequences representing usage scenarios for a given class. Although the discussion focuses on the analysis of Java programs, it is worth mentioning that our technique and *Scenariographer* are not specific to Java; we can use *Scenariographer* to analyze object-oriented software written in other programming languages.

Scenariographer consists of three major subsystems: **data gathering**, **repository**, and **analysis**. Figure 1 illustrates the high-level architecture of *Scenariographer*.

The **data gathering** subsystem collects the runtime information that is required to construct symbolic expressions representing the method invocation sequences for the objects of a given class. This information includes the unique identifier of an object, its type, and method entry and exit events. To collect the information, we use JVPROF, a dynamic analyzer that is implemented using the JVMDI and JVMPI interfaces [24, 25]. The runtime information collected by JVPROF is stored as an XML document, which is later exported to the repository.

The **repository** subsystem defines the data model and stores runtime information collected by the data gathering subsystem, such as program entities, relationships, and runtime events. The repository is manipulated using SQL and is queried using either SQL or SMQL (Software Modelling Query Language), our own query langauge [22]. The repository uses any JDBC-compliant database.

SMQL simplifies the data retrieval and analysis of program data to create software views. Even though the repository can be queried using SQL, designing queries for comprehending software systems using SQL is cumbersome. Many of the queries that are of interest, for example queries that involve the transitive closure of a relation, are not supported directly by SQL. SMQL is a set-based language that facilitates the definition of queries about entities, relations, and events by translating the SMQL code into SQL query statements. SMQL provides a built-in `closure` function as well as binary operators such as union, intersection, and join. Further details about the data gathering and repository subsystems are described elsewhere [22, 23].

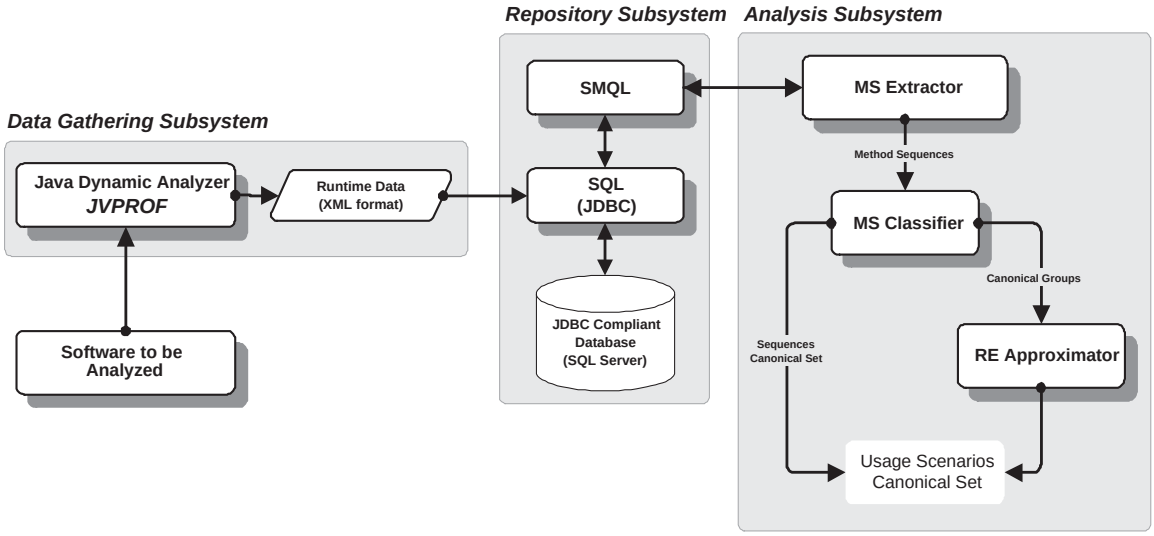

Figure 1. High-level architecture of Scenariographer

The **analysis** subsystem creates an object-interaction model from the runtime information, computes the canonical set, and identifies the various class usage scenarios. It consists of three components: MS Extractor, MS Classifier, and RE Approximator.

The *MS Extractor* (Method Sequence Extractor) creates the object-interaction model and extracts the method invocation sequences for a given class [22]. Objects are uniquely identified by their runtime references, and thus each object has its own method sequence. The MS Extractor then constructs symbolic representations of the method invocation sequences, which are use by the MS Classifier to compute the canonical set.

The *MS Classifier* (Method Sequence Classifier) is the component that implements the algorithm for computing the canonical set, and uses the resulting elements to create the canonical groups of method invocation sequences that correspond to class usage scenarios. The MS Classifier is implemented using MatLab and Perl scripts. The algorithms for computing the canonical sets and groups are described in detail in Section 4.

The *RE Approximator* (Regular Expression Approximator) is an optional component whose objective is to encode the elements of the canonical groups, and hence the sequences that define the class usage scenarios, into an expression that improves readability for the end user. The approximation process is described in more detail in Section 4.3.

To illustrate the relationship between method invocation sequences, canonical sequences, and canonical groups, consider the following example of the `File` class, with the symbols O, R, W, C representing the methods `open()`, `read()`, `write()`, and `close()`, respectively.

Assume that the method invocation sequences obtained from the execution of a program that contains the `File` class are as follows:

$$\begin{aligned}\mathcal{P} \;=\; & \{ORC, ORRC, ORRRC, ORRRRC, OWC,\\ & OWWWRC, ORWWWWWC,\\ & OWWWWWWWWWWC, ORRRRRRRRRRWC\}\end{aligned}$$

Given the above sequences, the MS Classifier will produce the canonical set:

$$\mathcal{P}' \;=\; \{ORRC, OWWWRC\}$$

and the canonical groups:

$$\begin{aligned}P_1 \;=\; & \{ORC, ORRC, ORRRC, ORRRRC,\\ & ORRRRRRRRRRWC\}\\ P_2 \;=\; & \{OWWWRC, OWWWWWWWWWWC,\\ & ORWWWWWC, OWC\}.\end{aligned}$$

Note that each canonical group is centered around an element of $\mathcal{P}'$.

To improve readability, these canonical groups can be simplified to regular expressions (as described in Section 4.3) as follows:

$$\begin{aligned}P_1 \;&=\; O(R)^{1,2,3,4,10}(W)^{0,1}C\\ &\approx\; O(R)^{+}(W)^{?}C\\ P_2 \;&=\; O(R)^{0,1}(W)^{1,3,5,10}C\\ &\approx\; O(R)^{?}(W)^{+}C\end{aligned}$$

The powers indicates the periodicity of the symbol. For example a periodicity of $(0, 1)$ means zero or one (optional)

occurrence, $(1, 3, 5)$ means one, three or five occurrences of the symbol. The first group P_1 represent a usage scenario that can be summarized as the file-reads scenario, and the group P_2 represent the file-writes scenarios.

4 Computing canonical sets and class usage scenarios

This section describes the algorithms for computing canonical sets and classifying the method invocation sequences into groups that represent class usage scenarios.

Section 4.1 describes the algorithm for computing canonical sets. The description is at a high-level; for a detailed description of the algorithm please refer to [10, 11].

Section 4.2 describes how the elements of the canonical set are used to form canonical groups of method invocations sequences.

4.1 Computing canonical sets

Intuitively, for a given set of method sequences under a known similarity function, its canonical set is the small subset of its members that best characterizes the elements of this set. Formally, if $\mathcal{P} = \{p_1, ..., p_n\}$ is the set of method invocation sequences and $\mathcal{S} : p \times p \rightarrow \mathbb{R}^{\geq 0}$ denotes the similarity function, we are interested in a *small* subset $\mathcal{P}' \subseteq \mathcal{P}$ that maximizes the similarity between $\mathcal{P}'$ and $\mathcal{P} \setminus \mathcal{P}'$. We use the set notation $\mathcal{P} \setminus \mathcal{P}'$ to denote our original set $\mathcal{P}$ with the members of the canonical set $\mathcal{P}'$ removed. Unfortunately, problems with combinatorial constraints of this type are known to be computationally hard [13]. However, there is credible theoretical and experimental evidence [27] that good approximation frameworks can be developed to deal with such problems.

In recent work [10] we developed a framework, which is not domain-specific, for computing approximate solutions to the canonical set problem. The framework formulates the canonical set problem as an integer programming optimization with a flexible set of objectives and constraints. To address the intractability of the integer programming problem, we developed an original relaxation of the semidefinite optimization problem (SDP), following the approach of other researchers [15, 14, 19]. We then developed a procedure for constructing canonical sets from the solution of the relaxed semidefinite optimization.

Additionally [11], we introduced the notion of a *bounded canonical set*, BCS, which placed upper and lower bounds on the cardinality of the canonical set, removed the multi-objective formulation, and allowed for a total similarity function (*i.e.*, any two elements are comparable).

At a high level, the process of constructing the canonical set can be described as follows :

1. Encode the method sequences as strings.
2. Compute the similarity matrix using edit-distance.
3. Formulate the integer programming problem.
4. Perform a relaxation.
5. Use SDP to solve the relaxation.
6. Use rounding to obtain an approximate solution.

In the following sections we present an overview of the similarity measure for method sequences and our method for constructing bounded canonical sets.

4.1.1 Problem formulation

The BCS problem can be stated formally as follows: Given a set of method invocation sequences $\mathcal{P} = \{p_1, ..., p_n\}$, a similarity function $\mathcal{S} : p \times p \rightarrow \mathbb{R}^{\geq 0}$, and two integer bounds k_{min} and k_{max}, the *bounded canonical set* for $\mathcal{P}$ is a subset $\mathcal{P}' \subseteq \mathcal{P}$ that best characterizes the elements of $\mathcal{P}$ with respect to the similarity function $\mathcal{S}$ while having cardinality k, where $k_{min} \leq k \leq k_{max}$. If it is unclear how to set the k_{min} and k_{max} parameters, then k_{min} can be set to 1 and k_{max} to n, the algorithm will then try to obtain a solution based on these inputs.

4.1.2 Similarity measure

We now present an overview of the distance function used to measure the similarity between method invocation sequences.

As our initial distance function we selected the well-known *edit-distance* [18]. Intuitively, the edit-distance between two method invocation sequences p_i and p_j measures the minimum cost of changes needed to transform p_i to p_j. The set of valid changes and operations includes, *insertions*, *deletions*, and *substitutions* in p_i or p_j.

Associated with each valid operation there exists a cost. The cost represents the amount of a single modification required to transform one sequence to another. The cost is not necessarily a uniform function for different kinds of operations. For example, inserting a method invocation which does not already exist in the sequence is a more costly operation than inserting a call that already exists, as it increases the potential of resulting in a different usage scenario. The cost is a function of the frequency of a particular method invocation in the sequence. For example, inserting a method invocation represented by the symbol R into the sequence $< ORRC >$ will cost more than inserting it into the sequence $< ORRRRRRRC >$, because the sequence $< ORRC >$ has fewer Rs. Furthermore, the cost is a function of the location. For example, inserting an R after the first R in the sequence $< ORRWWC >$ is cheaper than

inserting the R after the first W. The cost function, which accounts for both the frequency distribution of invocations in a sequence and their location, is a windowed inverse exponential function. A closed form description of the cost function can be found in [4]. The cost of a sequence of operations is defined as the sum of the individual costs.

The goal is to find a sequence of changes with minimum cost to transform p_i into p_j. It is known that the problem of finding the minimum cost of edit sequences to transform p_i into p_j can be solved in polynomial time using an efficient dynamic programming algorithm [2].

The edit-distance is used to create a similarity matrix that holds the similarity between all pairs of strings (method sequences). We define the similarity as:

$$S_{ij} = \frac{1}{1 + d(p_i, p_j)}$$

where $S_{i,j}$ is the similarity between string p_i and p_j, and $d(p_i, p_j)$ is the distance between p_i and p_j calculated by the edit-distance function.

We would like to note that functions to measure similarity between method invocation sequences are not limited to edit-distance and the cost functions described above. It is possible to define other functions, perhaps even more appropriate for the type of applications we are interested to analyze. This is the topic of ongoing research and we plan to present results that compare the performance of different similarity functions in future work.

4.1.3 BCS construction

In this section, we describe our method for constructing bounded canonical sets in polynomial time. Starting with a set of method invocation sequences $\mathcal{P} = \{p_1, ..., p_n\}$ of a class, and a similarity function $\mathcal{S} : p \times p \rightarrow \mathbb{R}^{\geq 0}$, we construct a complete edge weighted graph $G = G(\mathcal{P})$, where the method invocation sequences $\{p_1, ..., p_n\}$ are represented by vertices, and the edges have weights corresponding to the measure of similarity between the vertices. We use V to denote the vertices of G and V^* to denote the subset of V corresponding to the BCS.

Examining the canonical set shown in Figure 2, we categorize the edge set of G into three groups: *intra* edges, where both endpoints are within the canonical set V^*, *cut* edges, where one of the endpoints is in the canonical set and the other is not, and *extra* edges, which are the rest. Our goal is to minimize the sum of the weights of the intra edges, while at the same time maximizing the sum of the weights of the cut edges. In doing so, we attempt to place the vertices that are most representative of the others in the BCS, while keeping the BCS members as dissimilar as possible.

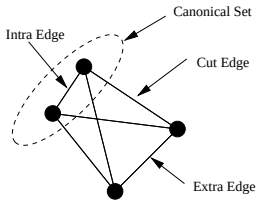

Figure 2. BCS edges

The problem of maximizing the weight of the cut edges is known to be NP hard. Goemans and Williamson [15] explored the problem MAX-CUT in graphs, and used semidefinite programming (SDP) relaxations to provide good approximate solutions. See Goemans [14] and Mahajan and Ramesh [19] for a survey of recent results and applications of SDP. Following their lead, we formulate our problem of BCS as an integer programming problem, and then use SDP to give us a good approximate solution.

4.2 Computing canonical groups

Given $\mathcal{P} = \{p_1, ..., p_n\}$, the set of method invocation sequences, and $\mathcal{P}' = \{q_1, ..., q_k\}$, the set of canonical method invocation sequences, the MS Classifier uses a minimum distance classification approach to compute the canonical groups. The steps used to compute the canonical groups are as follows:

1. For each method invocation sequence, p_i, in $\mathcal{P}$, compute its similarity to all of the canonical sequences in $\mathcal{P}'$.
2. Place the method invocation sequence p_i in a group that corresponds to the canonical method invocation sequence, q_i, to which it is most similar (*i.e.*, smallest edit distance measure).
3. In the event of a tie in max similarity, place p_i in all of the corresponding canonical groups.

Each resulting canonical group contains similar method invocation sequences that represent a usage scenario of the class.

4.3 Regular expression approximation

As we mentioned earlier, the Regular Expression Approximator (*RE Approximator*) is an *optional component* whose objective is to encode the elements of the canonical groups into an expression that improves readability for the end user. In what follows in this section we describe this approximation process.

Our implementation is based on work in the areas of string matching [20, 6] and the approximation of regular

$(A)^1$	$(BIKLMH)^1$	$(CD)^1$	$(K)^1$	$(EFGH)^{45}$	ϵ
$(A)^1$	ϵ	$(CD)^1$	ϵ	$(EFGH)^{13}$	$(J)^1$
$(A)^1$	$(BIKLMH)^{1,0}$	$(CD)^1$	$(K)^{0,1}$	$(EFGH)^{13,45}$	$(J)^{0,1}$
(A)	$(BIKLMH)^?$	(CD)	$(K)^?$	$(EFGH)^+$	$(J)^?$

Table 1. Alignment of subsequences

expressions [9]. The outline of the approximation process is as follows:

1. Compute the longest common subsequence (LCS) of each method invocation sequence. The LCS sets of all method invocation sequences are combined into one LCS set, which is used in the next step to ensure that the segmentation of all method invocation sequences is performed in the same order for all sequences.

2. Compress the method invocation sequences using the combined LCS set. This step segments each method invocation sequence into subsequences and finds the repetitions of each subsequence. Note that each subsequence is a member of the LCS set. For example, the following two sequences:

$$ABIKLMHCDKEFGHEFGH\cdots EFGH$$
$$ACDEFGHEFGHEFGHEFGH\cdots EFGHJ$$

Can be compressed into:

$$(A)^1(BIKLMH)^1(CD)^1(K)^1(EFGH)^{45}$$
$$(A)^1(CD)^1(EFGH)^{13}(J)^1$$

3. Align the compressed sequences to produce a single expression that represents all of the method invocation sequences. The alignment is performed using the anchor subsequences, which are the common repeated subsequences. Empty (ϵ) subsequences are inserted as needed to produce the aligned sequences. For the above example, the alignment process produces the alignment shown in the first two rows of Table 1. Then the aligned sequences are combined into a single expression as shown in the third row of Table 1.

4. Approximate the aligned single expression as a regular expression using the following rules:

   - If the subsequence has a minimum periodicity of zero and a maximum periodicity of one, then the subsequence is optional ?(zero or one).
   - If the subsequence has a monotonically increasing periodicity starting at zero, then set the periodicity to $*$ (zero or more).
   - If the subsequence has a monotonically increasing periodicity starting at one, then set the periodicity to $+$ (one or more).

   The approximated regular expression is shown in the fourth row of Table 1.

The approximated regular expressions of the method sequences represent the usage scenarios of the class under study.

Program	classes [1]	methods[2]	objects	events
Jext	257/426	983/4,495	54,434	181,964
Jetty	102/189	899/2,088	1,459	24,820

(1) The number of classes exercised versus the total number of classes in the source code

(2) The number of methods exercised versus the total number of methods in the source code

Table 2. Systems analyzed

5 Case study

To evaluate our approach, we used *Scenariographer* to generate usage scenarios for two widely-used classes: `gnu.regexp.RE` and `java.net.Socket`. Both classes are used in open source programs written in Java, which gave us access to the source code of applications that are of reasonable size for our evaluation. The class `gnu.regexp.RE` is used by the text editor `Jext` [1], while the `java.net.Socket` class is used by the `Jetty` web server [21]. Relevant properties of these applications are shown in Table 2.

An outline of the process used to infer the class usage scenarios for both classes is as follows:

- We executed the Java application in profiled-mode using the JVPROF profiler. We exercised selected features of the application and the collected runtime events were stored in the repository.
- We used the method sequence extractor (*MS Extractor*) to extract the method invocation sequences from the class instances. The *MS Extractor* eliminated all self-calls (*i.e.,* invocations an object made to itself) from the method sequences to keep only the public methods that were invoked by other objects.
- The method sequence classifier (*MS Classifier*) was then used to compute the canonical sets of method

invocation sequences and corresponding canonical groups.

- On the last step, we exercised the option to combine the method invocation sequences of each group into a single regular expression by executing the *RE Approximator*.

The `gnu.regexp.RE` class, which is part of the GNU regular expression package, provides an interface for compiling and matching regular expressions. The analysis of the method sequences of the `RE` class produced three usage scenarios. The association between symbols and methods for the `RE` class is as follows:

A	$<$ init $>$	Constructor
B	match	
C	isMatch	
D	chain	

The simplified sequences of the groups that represent each scenario are:

- Group-1:

$$\{A(B)^{178}, AB, A(B)^2, A(B)^7, A(B)^{18}, A(B)^{31}\}$$

The combined expression of the above sequences is:

$$A(B)^{1,2,7,18,31,178} \approx A(B)^+$$

This group represents a usage scenario in which the `RE` object is created (<init> method), then one or more `match` methods are invoked. The `match` method checks if the input, in its entirety, is an exact match of a regular expression. The scope of the `match` method is the `gnu.regexp` package. The `ismatch` method is essentially similar to the `match` method with the exception of its public scope, which makes it visible outside of the `gnu.regexp` package.

- Group-2:

$$\{AC, A, AB, A(C)^2, A(C)^4, A(C)^7, A(C)^{14}, A(C)^{22}, A(C)^{160}, AD\}$$

The combined expression is:

$$A(C|B|D)^{0,1}(C)^{0,2,4,7,14,22,160} \approx A(C|B|D)^?(C)^*$$

This group represents a usage scenario in which the `RE` object is created (<init> method), then an optional invocation to one of the {`match`, `isMatch`, `chain`} methods, followed by zero or more invocations of the `isMatch` method. Note that the `chain` method is used to add a token that corresponds to part of a regular expression to the internal `RE` list data structure.

- Group-3:

$$\{AD(B)^{178}, AD, ADB, AD(B)^2, AD(B)^7, AD(B)^{31}, AD(B)^{18}\}$$

The combined expression is:

$$AD(B)^{0,1,2,7,18,31,178} \approx AD(B)^*$$

This group represents a usage scenario in which the `RE` object is created (<init> method) followed by an invocation to the `chain` method, followed by one or more invocations to the `isMatch` method.

It is worth noting that group-1 and group-3 could be identified as one scenario of the form $A(D)^?(B)^*$.

Our second example illustrates that for certain classes, such as `java.net.Socket`, there are multiple practical issues that one has to deal with to determine usage scenarios. For example, we found out that method invocation sequences can get long and complex, and may contain repeated subsequences, which may not be traceable to the developer's source code.

Consider the following method invocation sequence from the `Socket` class, which we obtained from the `Jetty` web server:

$$AEHDEFGHNC(EFGH)^2(NEFGH) \hookleftarrow (N(EFGH)^2N(EFGH)^2N(EFGH)^2(NEFGH))^2 \hookleftarrow ((EFGHN)(EFGH)^2(NEFGH))^2$$

Each symbol in the above sequence represents an invocation to one of the methods shown below:

A	<init>	*B*	setImpl
C	getInputStream	*D*	getOutputStream
E	getLocalAddress	*F*	getLocalPort
G	getPort	*H*	getInetAddress
J	close	*I*	postAccept
K	setSoTimeout	*L*	setSoLinger
M	setTcpNoDelay	*N*	getSoTimeout
O	connect	*P*	isClosed

In the above sequence, we observe the following repeated subsequences: $\{EFGH, NEFGH, EFGHN\}$. We inspected the source code and found that these method subsequences are not traceable to the source code of the application that uses the `Socket` class. Rather, they are the result of invocations from the methods `SocketInputStream.write()` and `SocketOutputStream.read()`. This is a case where a class usage scenario involves a group of classes (*i.e.*,`java.net.Socket`, `SocketInputStream`, and `SocketOutputStream`).

We were interested to find out if by grouping the calls from `SocketInputStream` and

`SocketOutputStream` we could improve the result. Indeed, by considering these classes as a group we were able to produce the simpler sequence:

$$\cdots (W)^3(RW)^2R(WR(W)^2RWR)^4$$

where,

R	java.net.SocketInputStream.read()
W	java.net.SocketOutputStream.write()

We see that the subsequences $\{EFGH,\ NEFGH,\ EFGHN\}$, in the `Socket`-only case, are replaced by simpler subsequences $\{W, R\}$.

The `Socket` example illustrates that, for certain classes, the usage scenarios do not reflect how a given class is used, unless its collaborating objects are incorporated in the analysis. We are investigating the estimation of usage scenarios for subsystems comprised of closely collaborating objects. Specifically, in future work we plan to highlight how sets of classes are used by treating them as a single subsystem. In the case of the `Socket` class, this set of classes will be the `Socket`, the `Input Stream`, and the `OutputSream` classes.

6 Conclusions and future work

In this paper we described an algorithm and a tool to estimate the usage scenarios of a class from its execution profile. The estimation process produces canonical groups, where each group comprises a set of similar method invocation sequences that represent a usage scenario. Each group of method invocation sequences can be further simplified into a regular expression.

Through a case study, we demonstrated the ability of the tool to collect data at run time, extract method invocation sequences from class instances, and classify the sequences into similar groups. The examples in the case study show how the estimation of usage scenarios simplifies the task of understanding how a system uses a class.

Our immediate research efforts will focus on three areas. First, the application and evaluation of different similarity cost functions to minimize the effect of overlapping groups. Second, the generalization of the usage scenarios to subsystems (*i.e.*, sets of collaborating objects). Third, the improvement of the usage scenario presentation by automatically generating pseudo code fragments similar to those found in textbooks and user documentation.

Our long term objective is to develop a tool capable of creating sample programs that illustrate different usage scenarios for a given class. *Scenariograph* provides us with some of the important capabilities that we need to develop and implement such a tool.

References

[1] *Jext: Source code editor*. http://www.jext.org/.

[2] Levenshtein distance: calculating similarity of strings. http://perlmonks.thepen.com/Levenshtein distance: calculating similarity of strings.html.

[3] G. Ammons, R. Bodik, and J. Larus. Mining specifcations. In *Proc. 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 4–16, Portland, OR., Jan 2002. ACM Press.

[4] A. Aner and J. R. Kender. A unified memory-based approach to cut, dissolve, key frame and scene analysis. In *Proc. Inernational Conference on Image Processing*, Greece, October 2001.

[5] D. Angluin and C. Smith. Inductive inference: theory and methods. *ACM Computing Surveys*, 15(3):237–269, September 1983.

[6] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial algorithms on words*. Springer-Verlag, 1985.

[7] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Computers*, 21:592–597, June 1972.

[8] A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Trans. Software Engineering*, SE-2:141–153, 1976.

[9] A. Brazma. Efficient algorithm for learning simple regular expressions from noisy examples. In *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence*, pages 260–271. Sringer-Verlag, New York, NY, 1994.

[10] T. Denton, J. Abrahamson, and A. Shokoufandeh. Approximation of canonical sets and their application to 2d view simplification. In *IEEE Conference on Computer Vision and Pattern Recognition*, Washington, DC, June 2004.

[11] T. Denton, M. F. Demirci, J. Abrahamson, A. Shokoufandeh, and S. Dickinson. Bounded canonical sets and their applications to view indexing. In *17th IAPR International Conference on Pattern Recognition*, Cambridge, United Kingdom, August 2004.

[12] M. Fowler and K. Scott. *UML Distilled: a brief guide to the standard object modeling language*. Addison-Wesley, 2nd edition, 2000.

[13] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Co., Baltimore, 1979.

[14] M. X. Goemans. Semidefinite programming in combinatorial optimization. *Mathematical Programming*, 79:143–161, 1997.

[15] M. X. Goemans and D. P. Williamson. .878-approximation algorithms for MAX CUT and MAX 2SAT. In *Twenty-sixth Annual ACM Symposium on Theory of Computing*, pages 422–431, New York, 1994.

[16] E. M. Gold. Language identification in the limit. *Inf. Control*, 10:447–474, 1967.

[17] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience*, 24(7):643–658, July 1994.

[18] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Daklady*, 10:707–710, 1966.

[19] S. Mahajan and H. Ramesh. Derandomizing approximation algorithms based on semidefinite programming. *SIAM Journal on Computing*, 28(5):1641–1663, 1999.

[20] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262 – 272, April 1976.

[21] Mort Bay Consulting. *Jetty Web Server and Servlet Container*. http://jetty.mortbay.org.

[22] M. Salah and S. Mancoridis. Toward an environment for comprehending distributed systems. In *Proceedings of Tenth Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, November 2003. IEEE.

[23] T. S. Souder, S. Mancoridis, and M. Salah. Form: A framework for creating views of program executions. In *International Conference on Software Maintenance*, 2001.

[24] Sun Microsystems, Inc. *Java Platform Debugger Architecture*.

[25] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI)*.

[26] T. Systa and K. Koskimies. Extracting state diagrams from legacy systems. In *Proc. 1997 ECOOP Workshops*, Lecture Notes in Computer Science (LNCS) 1357. Springer, 1997.

[27] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, Germany, second edition, 2003.