Identifying and Generating Missing Tests using Machine Learning on Execution Traces

1st Mark Utting
USC Business School
University of the Sunshine Coast
utting@usc.edu.au

2nd Bruno Legeard
Dept. DISC - FEMTO-ST
University of Bourgogne Franche-Comté
bruno.legeard@femto-st.fr

3rd Frédéric Dadeau
Dept. DISC - FEMTO-ST
University of Bourgogne Franche-Comté
frederic.dadeau@femto-st.fr

4th Frédéric Tamagnan
Orange Labs Services
Pessac, France
frederic.tamagnan@orange.com

5th Fabrice Bouquet
Dept. DISC - FEMTO-ST
University of Bourgogne Franche-Comté
fabrice.bouquet@femto-st.fr

***Abstract*—Testing IT systems has become a major bottleneck for many companies. Besides the growing complexity of such systems, shorter release cycles and increasing quality requirements have led to increased verification and validation costs. However, analysis of existing testing procedures reveals that not all artifacts are exploited to tame this cost increase. In particular, customer traces are usually ignored by validation engineers. In this paper, we use machine learning from execution traces (both customer traces and test execution traces) to identify test needs and to generate new tests in the context of web services and API testing. Log files of customer traces are split into smaller traces (user sessions) then encoded into Pandas DataFrames for data analysis and machine learning. Clustering algorithms are used to analyse the customer traces and compare them with existing system tests, and machine learning models are used to generate missing tests in the desired clusters. The tool-set is implemented in an open-source library called Agilkia.**

***Index Terms*—automated regression testing, machine learning, customer traces, clustering, test generation**

I. Introduction

The rapid development of agile practices in recent years, and now DevOps, i.e. continuous integration, testing and deployment, has increased the need for test automation. In the latest State of Testing 2019 survey, 87% of participants indicated that they work in an Agile context, 38% in a DevOps context [1]. This acceleration of the pace of releases increases the effort to develop and maintain automated test scripts. In the same survey, 74% of respondents reported using test automation, and 21% of respondents relied on test automation for more than 50% of their test needs. Efforts to automate tests are increasingly creating a project bottleneck: both in terms of the effort and time required and in terms of the availability of the required resources and skills. In 2018, Izzy Azeri, founder of 'mabl', an AI-driven testing start-up, noticed: *"As we met with hundreds of software teams, we latched on to this idea that developing — the process of writing new code and integrating it with your code bases — is very fast now, but there's a bottleneck in QA. Every time you make a change to your product, you have to test this change or build test automation."*

This increasing reliance on automated testing motivates the research work presented in this paper, which aims to use execution trace data (from both operational usage and test execution) to create and maintain automated tests. We focus on functional testing, at the system level, for black box testing of APIs or Web service calls. The input execution traces are in the form of logs, which are very often present in web applications, especially for debugging and performance monitoring purposes. Machine learning on these traces allows us to identify regression testing needs that are not met, and to generate new automated tests by automatic generation from a learnt model. This research is part of an active field in software testing that aims at facilitating test automation through machine learning. This is reflected both in the research work we summarize in the related work section and in the emergence of start-ups in the field such as Functionize, mabl, test.ai or testsigma, to name but a few.

Paper contributions

Exploiting user traces for software testing has been an important research focus for at least two decades, especially for web applications [2]–[4]. These research studies focused mainly on the construction of models of different kinds – FSM/eFSM, statistical, event graph [5], to apply various search techniques and algorithms for test generation, selection and prioritization. As stated by Arcuri [6], despite strong academia effort and intensive research work, these techniques are used rarely in industry, because of the complexity of using symbolic model-based approaches for software engineers. Our research work aimed at using machine learning models, learned from available software development and operational data, to provide test need identification and test generation services. The main contributions of this paper are:

1) Identifying regression test needs by comparing test execution traces and operational execution traces using clustering and visualization techniques.

2) Automating test generation using a predictive machine learning model of user traces to propose new test cases covering the identified regression test needs.
3) An open-source toolbox supporting these services [7].
4) Experimental evaluation on two industry web services.

In the rest of this paper, we start in Section II by presenting the related state of the art, then we describe in detail the techniques we use for trace analysis and clustering in Section III, for identifying test needs by comparing and visualizing clusters of usage traces and test execution traces in Section IV, and for generating new tests from a predictive machine learning model in Section V. This technical presentation is based on a running example which is introduced in Section III. Then we present and discuss the results on two case studies in an industrial context (Bus system and Supply chain) and discuss the validity and reproducibility of the work, in connection with the open-source implementation of the Agilkia toolbox [7].

II. Related Work

Over the years, there have been many papers that tackled trace analysis, mostly for anomaly detection, but also for test generation purposes. Most are based on symbolic techniques such as inferring FSM models from logs. In this related work section, we focus on using machine learning techniques.

A. ML-based Trace Analysis

We can separate this work into two groups: statistical Machine Learning (ML) approaches [8], [9] and Deep Learning approaches including Recurrent Neural Networks (RNN) and advanced Natural Language Processing (NLP) techniques based on log mining [10]–[14]. Logs record runtime information to monitor test and system execution traces, for example to analyze anomalous behaviors and errors. Due to the unstructured nature of logs, a first step is to parse logs into structured data, then encode them for machine learning.

For statistical ML approaches, encoding transforms each trace into a vector of event counts (e.g. bag-of-words). Lou *et al.* [9] mine invariants from these event counts, as these can exhibit workflow patterns in the program. Xu *et al.* [8] use Principal Component Analysis (PCA) to project traces into a reduced space and thus proceed to anomaly detection, analyzing correlations between traces within a sequence.

Given the sequential nature of traces, and their high numbers of string-valued parameters, NLP deep learning techniques are useful as well. Bertero *et al.* [11] consider the logs as plain text documents and use a word embeddings approach, following the framework of *word2vec* [10] to encode the data. Brown *et al.* [12] encode the traces with a bidirectional LSTM language model, stacking LSTM layers to obtain latent vectors representing tokens sequences in a more abstract and synthetic way. Then they develop five attention mechanisms for modeling sequences more accurately and providing interpretability about correlations between features.

B. User Trace Clustering

The clustering of execution traces is a technique widely used in the field of Process Mining [15], [16]. The goal is to discover business processes from user traces and to generate accurate business process models. Clustering algorithms like K-means, MeanShift, Agglomerative Hierarchical Clustering, and Self-Organizing Maps are used, with appropriate feature selection and distance measures, to segment logs to facilitate process mining and create classes of homogeneous cases.

Clustering of execution traces could also be useful for predictive website management. For this purpose Lakshmi et al. [17] propose a method to cluster users based on their navigation data. After computing the degree of influence of webpages for each user, they use X-means and Farthest-First clustering algorithms to classify them. For the same goal, Anupama et al. [18], use Hierarchical Agglomerative Clustering (HAC) to cluster web-user sessions, based on a similarity distance between sessions, measuring if one is a subset of another. Another use case of clustering execution traces is to identify log templates. Clustering can help us to connect logs together. Itkin et al. [19] use HAC approach with a brute-force comparison between strings. They also try a NLP-based approach, with k-means applied to words embeddings. Those embeddings are made with TF-IDF and then factorized with singular value decomposition technique.

C. ML-driven Test generation using traces

Some recent work has focused on learning about usage traces to generate tests for mobile applications (usually Android). Li *et al.* [20] use deep learning to train a model to generate tests that increases the realism of GUI actions.

Santiago *et al.* [21] employ LSTM models to generate test flows based on a test flow specification language to ensure the validity of generated tests. The approach is limited to abstract test flow generation (meaning without test data and expected results). Guo et Lu [22] apply hierarchical clustering algorithm to select and optimize test cases produced from user sessions. In this approach, the authors apply reduction and selection techniques to clusters computed from user sessions.

Our approach differs by working at the web service API level, and by comparing user traces and test traces to identify test needs, before generating tests to meet this need.

III. Trace Analysis and Clustering

In this section, we first introduce a running example to illustrate our techniques, namely a supermarket scanner. Then, we discuss processing of the raw data, namely the customer traces, into useful traces, followed by techniques for visualizing and clustering those traces using the Agilkia toolbox.

A. Running example

A Supermarket Scanner (scanner for short) is a device that is able to read products barcodes and store them into a shopping list. It is used by supermarket customers to perform shopping with self-service checkout. The usage of the scanner is as follows. First the customer identifies themselves to the scanner board system to **unlock** a scanner for shopping. During the shopping phase, the customer may **scan** the barcode of a product to add it to the purchase list as they put it into their physical shopping basket. If the customer decides to put a product back on the shelf they can also **delete** that product from the purchase list. If an unknown product is scanned, the scanner still adds that barcode to the purchase list but marks it for later processing. When the shopping is over, the scanner interacts with a checkout machine, which may occasionally request a '*control check*'. A control check means that a cashier takes the scanner and re-scans the barcodes of products in the basket. If the control check does not detect any products that are missing from the purchase list, or if no control check is done, the purchase list is **transmit**ted to the checkout and the customer proceeds towards payment. Just before payment, if unknown barcodes were scanned during the shopping, a cashier is requested to manually **add** these items on the checkout before continuing to payment. More generally, at this step, the customer can still ask the cashier to **add** or **delete** items in the purchase list. Finally, the customer will **abandon** the scanner (hang it up on the scanner board), **pay** for their purchases, and leave the supermarket.

The system is built based on four entities: *products* gathered into a *product database*, manipulated by the *scanner* and the *checkout*. Products are seen through their barcodes, which uniquely identify a particular kind of product. This system considers customer scenarios that start with the unlocking of the scanner and end with payment at the checkout. Customers interact firstly with a scanner (the **unlock**, **scan**, **delete**, **transmit** and **abandon** actions), and then with a checkout (the **openSession**, **closeSession**, **add**, **remove** and **pay** actions).

The Scanner software exists in two versions: a Java implementation and a Web-based simulator[1] in which a set of customers walk around the supermarket and shop for products by interacting with scanners and checkouts. These simulated customer behaviors are generated by traversing a finite-state machine usage model of possible behaviors using a pseudo-random algorithm.

B. Loading and Splitting Traces

Figure 1 shows part of the raw CSV (comma-separated values) file that is captured by logging the operation calls to the API of the objects. Each line is composed of 5 elements: a timestamp, a session identifier, an object identifier, an operation name, the parameters, and the result value of the operation invocation. We write a small trace reading function in Python to read this CSV file and convert it into a single Agilkia 'Trace' object that contains a sequence of many 'Event' objects - one for each API interaction.

As many customers are using the system in parallel, and the log file contains many operation calls in chronological order, this trace actually contains many different intertwined 'sessions', for different customers. However, the second column in the log file specifies the customer session ID (here, a number between 40 and 43), so the first preprocessing step is to split this one long trace into a separate usage trace for each customer. We do this by calling the Agilkia method `with_traces_grouped_by("sessID")`, and obtain 4818 traces.

```
# timestamp, sessID, object, action, inputs, output
1570573649196, 41, scan3, abandon, [], 0
1570573649191, 42, scan1, transmit, [checkout0], 0
1570573649197, 42, scan1, abandon, [], 0
1570573649355, 43, scan1, unlock, [], 0
1570573649358, 42, checkout0, openSession, [], 0
1570573649996, 43, scan1, scan, [5410188006711], 0
1570573650366, 43, scan1, scan, [5410188006711], 0
1570573650366, 42, checkout0, add, [3570590109324], 0
1570573650389, 44, scan2, scan, [3046920010856], 0
1570573651369, 42, checkout0, closeSession, [], 0
1570573652376, 42, checkout0, pay, [68.27], 0
1570573655132, 40, scan0, scan, [7640164630021], -2
1570573656245, 44, scan2, scan, [3270190022534], 0
1570573656633, 43, scan1, scan, [3474377910724], 0
...
```

Fig. 1. Raw data of the system log file.

For the scanner example, each of the resulting customer session traces will contain a sequence of scanner events (scanner objects are designated by `scan`*N*), starting with an *unlock* operation, followed by scanning operations then a successful transmission to the checkout (resulting in code 0) and ending either with a successful payment on the same checkout (checkouts are designated by `checkout`*N*), or with an interrupted control check that detects unscanned products. One of the resulting customer session traces is shown on the left hand side of Fig. 2.

C. Visualization of Traces

From our experience with several case studies, we have found that it is helpful to be able to view traces in a more concise form than the full sequence of events, so that traces can be easily understood at a glance and compared with each other. For this reason, the default display of traces in Agilkia is one character per event. The test engineer can provide a custom mapping from event names to characters, or let Agilkia choose characters automatically from the event names. For the scanner case study we choose the custom mapping:

```
{'unlock':'u', 'scan':'.', 'delete':'d',
 'add':'+', 'transmit':'t', 'openSession':'o',
 'closeSession':'c', 'pay':'p', 'abandon':'a'}
```

With this mapping, the trace on the left hand side of Fig. 2 is displayed as: `u..d.tao+cp`.

D. Clustering of Traces

The goal of clustering is to identify subsets of traces, called clusters, that gather together similar customer behaviors. To achieve that, we employ the *MeanShift* algorithm [23] which computes clusters automatically (we use its default hyperparameters). However, this algorithm requires its input data to be vectorized, in order to compute distances between two traces. Our default feature-extraction method is the simple Bag of Words representation [24]. This abstracts each trace

[1]The simulator can be run at https://fdadeau.github.io/scanette/?simu, traces are shown in the browser console.

```
scan1, unlock, [], 0
scan1, scan, [3474377910724], 0
scan1, scan, [3046920010856], 0
scan1, delete, [3046920010856], 0
scan1, scan, [7640164630021], -2
scan1, transmit, [checkout2], 0
scan1, abandon, [], 0
checkout2, openSession, [], 0
checkout2, add, [7640164630021], 0
checkout2, closeSession, [], 0
checkout2, pay, [14.95], 0
```

abandon	1
add	1
closeSession	1
delete	1
openSession	1
pay	1
scan	3
transmit	1
unlock	1

Fig. 2. A customer trace and its Bag of Words vectorization

into a vector of integers, where each entry represents the number of occurrences of a given operation in the customer trace. This ignores event ordering, but is sufficiently expressive for the examples in this paper.

Example 0.1 (From Traces to Vectors): Consider the usage (customer) trace in Fig. 2, left column, which represents shopping for three products, with one deleted and one whose barcode is not recognized (the -2 output), followed by transmission to the checkout and the manual adding of the unrecognized product by the cashier before payment. The vectorized representation of this usage sequence is given by the vector provided in the right column.

Once the test traces are converted into vectors these can be processed by the MeanShift clustering algorithm [25].

Example 0.2 (Results on the Scanner Traces): We generated a trace of 65000+ steps using our scanner simulator, which were split into 4818 customer traces as explained before. When running the MeanShift algorithm on this set of traces, 13 clusters are generated. Table I summarizes the clusters and provides an informal description of their content.

A manual inspection of the clusters shows that most of them make sense, as they represent the different behaviors that have been implemented in the scanner simulator. However, some clusters, such as 9-11, could be merged with existing ones as they seem to activate the same pattern of behavior as other clusters, but just with longer tests. But from a testing point of view, it might make sense to test longer test sequences rather than just shorter sequences.

IV. Cluster Visualization and Test Need Identification

When we have both customer traces and test execution traces available, it is useful to compare these two sets of traces to gain insight into the test coverage. In particular, it is useful to know if there are common customer behaviors that are not well-covered by the system test suite.

We perform this comparison in two ways:

1) analyze which customer clusters have system tests;
2) visualize all the customer traces and the system tests in a common 2D space, to enable a visual comparison of the density of customer traces versus system tests.

Our scanner application has 30 manually designed system tests, which we capture as traces. Figure 3 shows the number

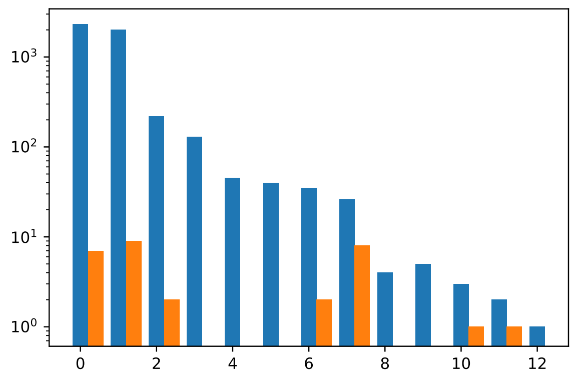

Fig. 3. Number of customer traces in each cluster (left/blue) compared with number of test traces (right/orange) in the same cluster. Y-axis is log scale.

of customer traces in each cluster, using a log scale for the y-axis so that smaller clusters are visible. The figure also shows the number of tests in each of those clusters (we classify each test trace into a cluster by using the same MeanShift model that was fitted to the customer traces). We can see that clusters 0, 1 and 7 are well-tested, clusters 2 and 6 each have two tests, and clusters 10 and 11 each have one test. The remaining clusters (3-5, 8-9 and 12) have no tests. This is because the system tests have a stereotyped shopping phase, and rarely combine scanning with deletion, so they have less variation than customer behavior.

Secondly, it is useful to give a visual comparison of customer traces versus system tests. To do this we use PCA (Principal Component Analysis) to map the raw attributes of each trace (normalized bag-of-words of the actions in the trace) into a 2D space. For our scanner example, PCA gives the following mapping to the X and Y dimensions (for readability, we elide dimensions with factors less than 0.04 here, even though they are included in the graph).

$$X = 0.69 * \#openSession + 0.19 * \#add + 0.69 * \#closeSession$$
$$Y = 0.73 * \#scan + 0.68 * \#transmission$$

The X dimension explains 88.76% of the variance in the trace data, and the Y dimension explains 5.7%, so the whole 2D graph visualizes 94.5% of the variation in the trace data. Figure 4 shows the customer traces. Note that they fall mostly into two vertical bands, because in customer traces the **open** and **close** operations typically both appear or both are absent.

In contrast, the system tests, visualized in Figure 5 using the same X,Y mapping, exhibit a wider range of X values. This is because the system tests are designed to test exceptional circumstances such as an **open** with no **close**, or multiple **close** actions. Overall, the visualization suggests that the system tests cover the customer traces reasonably well, except for the top few clusters and the bottom cluster (8). That is, clusters 3-5, 8-9 and 12 are missing from the system tests, as we saw in Fig. 3.

Id	Size	Description
0	2314	classical usage of the scanner (no control check, direct payment)
1	1996	shopping with unknown references that are added afterwards by the cashier before payment
2	218	shopping followed by a control check by the cashier then payment
3	129	similar to 2 with the addition of unknown products afterwards
4	45	similar to 3 but with longer sequences (more products to scan and control)
5	40	similar to 2 but with longer sequences (more products to scan and control)
6	35	same as 1 but with removals of products during the shopping
7	26	shopping, followed by a control which fails (detects a product that had not been scanned)
8	4	the sequences that were interrupted at the end of the log file
9	5	shopping, followed by a control, or not, followed by a manual addition of product by the cashier
10	3	short shopping with removal and direct payment without control
11	2	short shopping with removal and some manual additions of products
12	1	full complete sequence (shopping, control, addition of products)

TABLE I
RESULT OF THE CLUSTERING ON THE CUSTOMER TRACES

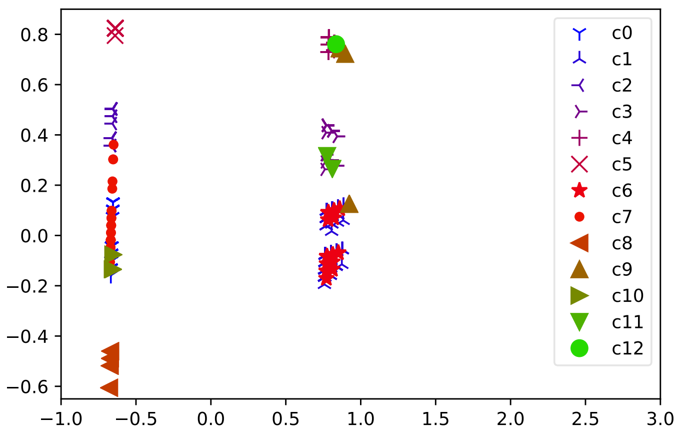

Fig. 4. Visualization of the 4818 customer traces in 13 clusters.

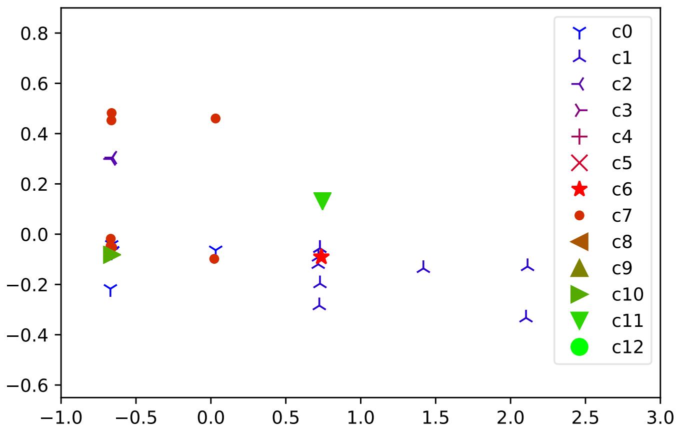

Fig. 5. Visualization of the 30 system test traces, mapped into the same 2D PCA space and clusters as the customer traces.

The Y mapping formula tells us that this must be because the tests in that vertical band do not call **scan** or **transmit** very often. Investigation of those system tests showed that they call **scan** at most 8 times (and **transmit** twice), which is far fewer times than some customer traces which scan up to 25 items. This suggests that it might be useful to add some system tests that scan larger numbers of items. This illustrates how the 2D visual comparison of customer traces versus system tests may identify areas that need further testing.

V. Test Generation Using a Predictive ML model

In this section we show how we can generate tests using machine learning models of customer behavior, with the goal of filling in areas of missing tests. For our running example, we illustrate this process on cluster 3 of the customer traces, which is the fourth most common customer behavior (129 traces, or 2.7% of the traces) but has no associated system tests. For example, the first 10 traces in this cluster are:

```
u.......t......tao+cp
u.......t......tao+cp
u.....t....tao+cp
u.....t....tao+cp
u.......t......tao+cp
u.....t...tao++cp
u....t...tao+cp
u.......t......tao+cp
u.....t....tao+cp
u.....t....tao+cp
```

We start by learning these sequences of actions, disregarding the parameter values initially. We generate all prefixes of all 129 traces, and then use the bag-of-words algorithm to encode each prefix into a numeric vector, and use this dataset as input for a multi-class supervised learning situation. The predicted output class is the next action in the sequence, or a special END token at the end of each sequence. For cluster 3, this gives 2621 prefixes in our training dataset, which is sufficient to learn a reasonably accurate model.

Table II shows the F1 score for several learning algorithms, calculated with 10-fold cross validation, with the variance shown in parentheses. Random Forests gives the best overall result, with Decision Trees, Gradient Boosting (GBC) and K-nearest Neighbors close behind. Dummy (scikit-learn DummyClassifier) is a base case classifier that makes a random choice of output class, so has low scores around $1/9$ in this scenario where there are nine possible output classes.

A. Generating User-like Behaviors

We can use any of these models with the Agilkia **SmartSequenceGenerator** class to generate test sequences that show realistic user-like behavior. Figure 6 shows a simplified version of the generation algorithm, which generates a `trace` of

Classifier	Cluster3	Cluster4	Cluster5
Tree	0.957 (0.026)	0.961 (0.051)	0.991 (0.051)
GBC	0.957 (0.026)	0.961 (0.051)	0.991 (0.051)
RandForest	0.957 (0.026)	0.966 (0.035)	0.996 (0.022)
AdaBoost	0.367 (0.000)	0.374 (0.006)	0.558 (0.135)
NeuralNet	0.934 (0.014)	0.947 (0.037)	0.999 (0.007)
KNeighbors	0.955 (0.017)	0.960 (0.042)	0.999 (0.007)
NaiveBayes	0.856 (0.022)	0.852 (0.029)	0.824 (0.015)
LinearSVC	0.899 (0.017)	0.852 (0.029)	0.827 (0.000)
LogReg	0.899 (0.019)	0.852 (0.029)	0.827 (0.000)
Dummy	0.112 (0.045)	0.117 (0.052)	0.156 (0.066)

TABLE II
F1 SCORE (WEIGHTED AVERAGE OF PRECISION AND RECALL) FOR MODELS LEARNED FROM CUSTOMER CLUSTERS 3-5.

```
trace = []
for i in range(length):
    [pa] = model.predict_proba(trace)
    [num] = rand.choices(range(len(pa)), pa)
    action = model.classes_[num]
    if action == "END":
        break
    else:
        trace.append(action)
```

Fig. 6. Algorithm to generate a random trace from a model that predicts the most likely next actions.

up to `length` events, given a `model` of customer traces and a random number generator `rand`. Note that it uses the `predict_proba` method that is supported by most classifiers to generate a probability distribution of all possible next actions that might occur (including the END of the trace), and then makes a random choice of next action according to those probabilities. This means that we can generate any desired number of random test traces.

For cluster 3, this generates traces like the following, which are quite accurate reflections of typical traces in cluster 3. These generated traces are quite homogeneous, because cluster 3 is homogeneous, but we shall see later that this technique of generating traces from learned models can generate heterogeneous traces when the training data is more varied.

```
u.....t....tao+cp
u.....t........tao+cp
u.......t.....tao++cp
u.....t........tao+cp
u.......t......tao+cp
```

B. Generating Systematic Test Suites

We can also use these learned models of customer behavior to systematically generate all 'common' sequences of events. This can be useful for generating a suite of tests that covers all the most common user behaviors.

The model has effectively learned a function from trace prefixes tr to probability distributions of the likely next events. By unrolling this model we obtain a tree of (tr, p) nodes, where tr is a trace prefix and p is the probability of that prefix (the product of the probabilities down that path of the tree). If tr ends with the END action, it is a leaf of the tree that corresponds to a complete test trace and its probability, whereas other nodes of the tree correspond to partial traces and the probability of all extensions of that prefix.

```
21.90% u.....tap
16.52% u.......tap
10.61% u.......tao+cp
10.07% u.....tao+cp
 5.23% u.............tao+cp
 3.72% u.............tap
 3.40% u.......tao++cp
 2.56% u.............tao++cp
 2.11% u.......t...tap
 1.61% u.....tao++cp
 1.53% u.............t.tap
 1.25% u....tap
 1.06% u.....t..ap
 1.01% u.............ttao+cp
82.57% of total behavior covered
```

Fig. 7. Systematic test suite generated from the whole Scanner customer model, including all traces with probability greater than 1.0%

Agilkia implements this algorithm via a depth-first recursive search that is similar to Fig. 6, but explores *all* possible next events with their probabilities. Given a maximum trace length L and minimum probability P, the search explores all prefixes up to depth L, skipping over any nodes whose probability is less than P, and returning all complete and partial traces that are found. Fig. 7 shows the resulting test suite when $L = 35$ and $P = 0.01$. Executing the first four of these generated tests means we would have tested 59.1% of the most common customer behaviors for the Scanner application, and executing the whole 14 tests would cover 82.57% of the behaviors.

VI. EXPERIENCE REPORT

This section describes the results of applying the above techniques to two industry case studies: 'Bus System' and 'Supply Chain'.

A. Bus System Case Study

Our first industry case study is a web service for tracking school buses and students. Each bus reports its GPS position to the server every minute, as well as other events such as students swiping their ID cards upon entering or exiting the bus, drivers recording absent students, etc. We analyzed anonymized traces from 15 buses recorded over one day, with a total of 3267 events and the following event frequencies:

```
SaveGPS '.'                          2808.0
SNSCheckIn 'i'                        133.0
SNSCheckOut 'o'                       122.0
SNSMarkAbsentWithLocation 'A'          68.0
Login 'L'                              30.0
GetSchoolManifestForRunType 'M'        30.0
ConfirmPreCheck 'P'                    30.0
SNSBulkCheckOut 'O'                    14.0
SNSBulkCheckIn 'I'                     14.0
GetContacts 'C'                         2.0
LoginOptions '?'                        1.0
```

Clustering: MeanShift chose five clusters. Four were single traces with unique features (e.g. the only trace with a LoginOptions event; the only trace with GetContacts events), and the remaining main cluster contained the other 11 traces which had more similar behavior. Choosing one trace from each cluster gives a small regression test suite.

Test Generation: The models learned from this small dataset had low F1 scores (0.379 for TreeClassifier, 0.375 for GBC, 0.365 for KNeighbours, etc.) because the number of adjacent GPS events was highly variable. But the resulting models were still able to generate realistic sequences of events (not shown here due to lack of space).

Systematic generation of the suite of *all* sequences with probability $P > 1\%$ and $length \leq 300$ took 6.7 seconds on a MacBook Pro (i5-4258U 2.40GHz) and gave the following suite of 11 tests.[2] These cover 33.75% of the observed behaviors, which is low because there are many lower-probability traces that vary only by the number of GPS events. In future work we plan to investigate abstraction of such repetitive sequences so that fewer, but more abstract, traces are generated, and they cover a higher percentage of behaviors.

```
 6.67% LP?M.ii.i.i.i.i.i.i.i.i.i.O.LPM.AA.I.(o.)^10
 5.56% LPM.i.i.i.i.i.i.O.LPM.I.o.o.o.o.o.o.
 4.44% LPMAAA.i.i.i.i.i.i.i.O.LPMAAA.A.A.I.(o.)^5
 4.44% LPMAAA.A.i.i.i.i.iO.LPM.AAAA.I.o.o.o.o.o.
 2.78% LPMA.C.i.i.i.i.i.i.i.i.i.O.LPMA.I.(o.)^6ooC.o.
 2.78% LPMA.i.i.i.i.i.i.i.O.LPM.I.o.o.o.o.o.o.
 1.67% LPMAA.A.i.i.i.i.i.i.i.O.LPMAAA.A.A.I.(o.)^5
 1.67% LPMAA.A.A.i.i.i.i.iO.LPM.AAAA.I.o.o.o.o.o.
 1.39% LPM.iiiA.i.i.i.i.i.i.i.i.i.O.LPMAI.(o.)^8ooo.o.
 1.25% LPMAA.i.i.ii.AAA.O.LPMAA.I.oo.ooooo.
 1.11% .LPM.i.i.i.i.i.i.O.LPM.I.o.o.o.o.o.o.
33.75% of total behavior covered
```

We executed these 11 generated test sequences on the Bus System web service, using Agilkia to send each event to the web service, and to retry the event up to 10 times until it was successful. Input values were chosen randomly from a separate data table for each named input, typically with 10% incorrect values so that error cases would be exercised.

Out of the total 2421 events in the generated tests, 2113 (87.3%) returned success when executed. The other 308 events (12.7%) returned an error status initially, but were successfully executed by retrying them a few times (average 1.3 times, maximum 6 times) with a new random choice of input values.

Overall, 8/11 of the event types were tested with both error and success results, while the remaining 3/11 (infrequent) event types were tested with only success results. This is a useful result, and these traces are now being adopted as automated regression tests by our industry partner. The traces have good coverage of the successful events, but it would be nice to have more systematic coverage of the error cases. In our ongoing work we are investigating use of machine learning to choose input values, and test generation algorithms for covering error cases more systematically.

B. Supply Chain Case Study

The second industry case we have worked on is a set of web services for managing maintenance equipment. For each repair job, a list of required equipment is created by a remote operator. Technicians use a mobile app to record when they collect the required equipment (**PrepareOrder**, **DeliveredOrder**) and when they return it (**EquipmentReturn**, **CloseOrder**). They are also able to create their own list of required equipment (**CreateCustomOrder**, **PrepareCustomOrder**) if a job requires it but the operator has not had the time to create it. The application calls a **StockConsistency** web service for reliability purposes, after each delivery of a regular order or when the database is unreachable, to inform the operator to do the stock transaction manually.

Id	Size	Description	Example
0	125	Creation of a custom order and then preparation and delivery.	NpD
1	85	Preparation and delivery of a regular order and then Stock Consistency call.	PD-
2	48	Same as cluster 1 but twice in a row.	NpDNpD
3	31	Equipment Return or Stock consistency call.	R
4	24	Sequences of cluster 2, then cluster 1.	PD-NpD
5	29	Same as cluster 2 but twice in a row.	NpDNpD

TABLE III
RESULT OF CLUSTERING FOR THE SUPPLY-CHAIN CASE STUDY.

We analyze two days traces from 437 sessions, with a total of 2898 events and the following event frequencies:

```
PrepareOrder         'P'   374.0
DeliveredOrder       'D'   720.0
EquipmentReturn      'R'   227.0
CloseOrder           'V'     5.0
CreateCustomOrder    'N'   421.0
PrepareCustomOrder   'p'   468.0
StockConsistency     '-'   681.0
CancelOrder          'X'     2.0
```

Clustering: MeanShift generates 51 clusters. The largest six clusters are described in Table III. They represent 78% of the sequences and correspond to the nominal cases. The remaining 45 clusters contain less than 10 sequences and correspond to anomaly cases, containing events like **CancelOrder**.

Test Generation: Using Agilkia, we generated a systematic test suite of all paths with $P > 1\%$, obtaining the following 10 test sequences:

```
24.26% NpD           15.23% PD-        7.55% NpDNpD
 3.89% R              2.19% PD-R       2.01% PD-PD-
 1.51% NpDNpDNpD      1.28% PD-NpD     1.07% NpDNpDR
 1.07% NpDR
60.05% total behavior covered
```

This systematic test suite covers well the behavior of our set of web services with a coverage ratio of 60.05%. Furthermore, this test suite matches well with our first six nominal clusters.

C. Experimental Validation, Validity and Replicability

We carried out an experimental validation of our approach at two levels: on a fully controlled application (the Scanner example) and on two industry case studies. There are two main research questions to be answered: **RQ1:** To what extent are the test needs identified by comparing clusters on test and execution traces valid? **RQ2:** To what extent do the tests generated by the predictive learning model and the percentage of coverage obtained allow us to meet the identified test need?

[2]The average length of each of these tests is 220.1 steps (std.dev.=20.3), so to be able to display them we compress all repeated SaveGPS events to a single '.' and write $[s]^n$ for a subsequence s that is repeated n times.

For the scanner example, we have a formal state machine model of the application. This allowed us to evaluate the effectiveness of the clustering by matching the set of transitions that were covered by the tests in each different cluster. We found that the clusters are coherent, because tests in each cluster cover a restricted set of paths through the state machine model, all leading to the same final state.

For the Bus and Supply-Chain applications, which are operational, the evaluation was carried out through discussions with the project teams, with a system developer and tester. In both case studies, the team did not have significant numbers of automated tests at the system level. The evaluation therefore focused firstly on the representativeness of the clusters, and secondly on the relevance of the generated tests. For both applications, the generated tests and associated probabilities were considered relevant for non-regression tests and good candidates for automation.

These results are still preliminary, but they are promising and replicable. All the Agilkia code is accessible as open source [7], and the data and Python scripts for the Scanner experiments are included in the `examples/scanner` folder of that GitHub repository (in branch `aitest2020`) so that these experiments can be reproduced.

VII. Conclusions

Comparing the execution traces of automated regression tests with the operational user execution traces enables us to identify clusters of usage patterns that are sufficiently or insufficiently tested in the regression testing. Learning models from some of those clusters of user traces allows us to generate missing tests based on the probabilities of the sequences, thus providing regression test coverage by mimicking end-user usage of the system.

All the developments made are available in open-source form on the Agilkia repository. Our current work is focused on improving the clustering configuration to increase the relevance of clusters. For test generation, we study the learning of test data equivalence classes on test execution traces to be able to concretize test scripts with relevant test data. Our objective is to provide a complete, open source toolbox, that uses machine learning to analyze user logs in various ways, and to generate missing automated regression tests.

VIII. Acknowledgement

This work was supported in part by the French National Research Agency: PHILAE project (N° ANR-18-CE25-0013).

References

[1] PractiTest, "State of testing survey 2019," 2019, available from https://qablog.practitest.com/state-of-testing/.

[2] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, "Leveraging user-session data to support web application testing," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 187–202, March 2005.

[3] S. Sampath, S. Sprenkle, E. Hill, L. Pollock, and A. Greenwald, "Applying concept analysis to user-session-based testing of web applications," *IEEE Trans. on Software Eng.*, vol. 33, pp. 643–658, 11 2007.

[4] P. Tonella, F. Ricca, and A. Marchetto, "Chapter 1 - recent advances in web testing," in *Advances in Computers*, ser. Advances in Computers, A. Memon, Ed. Elsevier, 2014, vol. 93, pp. 1 – 51.

[5] A. Andrews, A. Alhaddad, and S. Boukhris, "Black-box model-based regression testing of fail-safe behavior in web applications," *Journal of Systems and Software*, vol. 149, pp. 318 – 339, 2019.

[6] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, Aug 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9570-9

[7] "Agilkia toolbox," 2019, available from https://github.com/utting/agilkia.

[8] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. Big Sky, Montana, USA: ACM Press, 2009, p. 117.

[9] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining Invariants from Console Logs for System Problem Detection," in *USENIX annual technical conference*, 2009, pp. 24–37.

[10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, 2013, pp. 3111–3119.

[11] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection," in *2017 IEEE 28th Int. Symp. on Software Reliability Engineering (ISSRE)*. Toulouse: IEEE, Oct. 2017, pp. 351–360.

[12] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent Neural Network Attention Mechanisms for Interpretable System Log Anomaly Detection," *arXiv:1803.04967 [cs, stat]*, Mar. 2018, arXiv: 1803.04967.

[13] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. Dallas, Texas, USA: ACM Press, 2017, pp. 1285–1298.

[14] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 4739–4745.

[15] M. Song, C. Günther, and W. M. P. Aalst, "Trace clustering in process mining," in *Business Process Management Workshops, BPM 2008*, ser. Lecture Notes in Business Info. Proc., vol. 17, sep 2008, pp. 109–120.

[16] M. de Leoni, W. M. P. Aalst, and M. Dees, "A general process mining framework for correlating, predicting and clustering dynamic behavior based on event logs," *Information Systems*, vol. 56, 07 2015.

[17] P. Dhana Lakshmi, K. Ramani, and B. Eswara Reddy, "Efficient Techniques for Clustering of Users on Web Log Data," in *Computational Intelligence in Data Mining*, H. S. Behera and D. P. Mohapatra, Eds. Singapore: Springer Singapore, 2017, pp. 381–395.

[18] D. Anupama and S. D. Gowda, "Clustering of web user sessions to maintain occurrence of sequence in navigation pattern," *Procedia Computer Science*, vol. 58, pp. 558–564, 2015, second Int. Symp. on Computer Vision and the Internet (VisionNet'15).

[19] I. Itkin, A. Gromova, A. Sitnikov, D. Legchikov, E. Tsymbalov, R. Yavorskiy, A. Novikov, and K. Rudakov, "User-assisted log analysis for quality control of distributed Fintech applications," in *2019 IEEE Int. Conf. On Artificial Intelligence Testing (AITest)*, April 2019, pp. 45–51.

[20] Y. Li, Z. Yang, Y. Guo, and X. Chen, "A deep learning based approach to automated android app testing," *CoRR*, vol. abs/1901.02633, 2019. [Online]. Available: http://arxiv.org/abs/1901.02633

[21] D. Santiago, P. J. Clarke, P. Alt, and T. M. King, "Abstract flow learning for web application test generation," in *Proc. of 9th ACM SIGSOFT Int. Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: ACM, 2018, pp. 49–55.

[22] Y. Guo and L. Lu, "An automatic test case generation method based on user session and agglutinate hierarchical clustering algorithm," *Journal of Comp. Methods in Sciences and Eng.*, vol. 19, pp. 1–14, 10 2018.

[23] Yizong Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, Aug 1995.

[24] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[25] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Trans. on Information Theory*, vol. 21, no. 1, pp. 32–40, January 1975.