Software Quality Journal manuscript No.
(will be inserted by the editor)

Automatically Learning Usage Behavior and Generating Event Sequences for Black-Box Testing of Reactive Systems

M. Furkan Kıraç · Barış Aktemur · Hasan Sözer · Ceren Şahin Gebizli

Received: date / Accepted: date

Abstract We propose a novel technique based on recurrent artificial neural networks to generate test cases for black-box testing of reactive systems. We combine functional testing inputs that are automatically generated from a model together with manually-applied test cases for robustness testing. We use this combination to train a Long Short-Term Memory (LSTM) network. As a result, the network learns an implicit representation of the usage behavior that is liable to failures. We use this network to generate new event sequences as test cases. We applied our approach in the context of an industrial case study for the black-box testing of a Digital TV system. LSTM-generated test cases were able to reveal several faults, including critical ones, that were not detected with existing automated or manual testing activities. Our approach is complementary to model-based and exploratory testing, and the combined approach outperforms random testing in terms of both fault coverage and execution time.

Keywords test case generation, black-box testing, recurrent neural networks, long short-term memory networks, learning usage behavior

1 Introduction

Reactive systems [1] refer to an important class of systems that repeatedly react and respond to their environment. The size and complexity of software that is employed in these systems are continuously growing due to the increasing number and variety of interactions with the environment. For example,

M. Furkan Kıraç, Barış Aktemur, Hasan Sözer
Ozyegin University, İstanbul, Turkey
E-mail: {furkan.kirac, baris.aktemur, hasan.sozer}@ozyegin.edu.tr

Ceren Şahin Gebizli
Vestel Electronics, Manisa, Turkey
E-mail: ceren.sahin@vestel.com.tr

Digital TV (DTV) systems [51] currently support web browsing, on-demand streaming, home networking, and many other functionalities. As a result, comprehensively testing these systems becomes costly [34].

Test automation [7,47] is a typical approach to reduce costs. This involves the automation of a set of various activities such as the generation of test cases, execution of these cases on the system under test, and validation of the results. In this work, we focus on the problem of automated test case generation for reactive systems such as DTVs that are steered based on a set of input events. Hereby, each test case is composed of a sequence of remote controller key press events supplied to the system under test.

Model-based testing [42] is commonly used for black-box testing to automatically generate test cases. This technique relies on a test model that specifies possible usage scenarios of the system under test. The main drawback of this approach is the model creation process, which is manual and therefore error-prone and labor-intensive. There have been several techniques introduced to automatically generate or extend test models [39,43]; these techniques mostly explore the graphical user interface of a system to synthesize a model of it [39,43]. They do not exploit the knowledge and experience of human testers, which is the main motivation behind exploratory testing [53,29, 30]. We previously introduced an approach [16] to automatically refine a test model based on events recorded during exploratory testing activities. However, one has to define a mapping of recorded low level events to abstract states of the test model to apply the approach.

There have been several approaches [14,36,12] for leveraging existing test suites to generate new tests. Some of these approaches combine black-box and white-box testing to be able to infer a model of the application [14]. Then, new test cases can be generated based on this model. Learning-based testing [36] is purely black-box; however, it requires a formal specification of the requirements. Other black-box approaches [12] do not rely on any model. They just combine or modify existing test cases to generate new ones.

In this work, we focus on black-box robustness testing, where the goal is to find usage scenarios that trigger crash failures. Our work is motivated by the testing practices of DTVs employed at Vestel[1], which is one of the largest TV manufacturers in Europe. For testing, besides automated execution of test scripts, manual exploratory tests [29,30] are performed as a complementary approach. These exploratory tests are not based on a predefined set of test cases or check-lists, but are purely based on the practical knowledge, insight, heuristics, and experience of the technician conducting the test. Test technicians regularly reveal faults that are otherwise hard to detect using automated techniques. During an exploratory testing session, the technician basically uses a hand-held remote controller to send commands to the DTV just like a regular end-user. Technicians perform manual tests iteratively by learning about the product, planning the testing work to be done, designing and executing the tests, and finally, reporting the results.

[1] `http://www.vestel.com.tr`

The drawback of exploratory testing is its cost due to its manual and human-centric nature. Nevertheless, they are an indispensable part of testing, because critical faults are mostly detected during exploratory testing activities [16]. Experienced test technicians have the necessary insight and years of domain knowledge to detect these faults. However, their knowledge and experience are unfortunately not documented. Hence, our work is driven by the following question:

Can we capture the insight of experienced test technicians by automatically learning from the actions they perform during manual testing?

Our ultimate goal is to be able to automatically generate representative test cases as a result of this learning process, so that the cost of manual testing can be alleviated without compromising the effectiveness of exploratory testing. To this end, we propose a novel technique for test case generation by employing artificial neural networks. In particular, we create a Long Short-Term Memory (LSTM) network [25], which is proven to be effective in learning sequences [50] (see Section 3). LSTM networks have been successfully applied to various sequence prediction tasks [50]. However, to the best of our knowledge, they have not been utilized for test case generation before.

We applied our approach in the context of an industrial case study to generate test cases for a DTV system. The code-base of this system is relatively mature. So, it is challenging to find scenarios that trigger (new) crashes. Yet, we obtained promising results by identifying several of those. We first trained an LSTM network with test cases that are generated from a test model. We used this network to generate new test cases. Then, we extended the training data with test cases that are recorded during manual exploratory testing activities. We generated a new set of test cases after another training step performed with this data. We observed that test efficiency is improved in terms of the number of detected faults per unit of time. Our approach also outperformed random testing and model based testing, which are used as baseline methods in our evaluation. In addition, we were able to detect several critical faults, including new ones that were not detected with existing test suites or manual testing activities, and also known faults that the test engineers had not been able to reproduce.

In the next section, we explain our general approach. In Section 3, we provide background information on LSTM networks and their use for sequence modeling. We explain the industrial case study in Section 4. We provide the details of our approach and evaluate the results in Section 5. We summarize the related studies and position our work with respect to them in Section 6. Finally, we provide concluding remarks and future work directions in Section 7.

2 Approach

Throughout the paper we use the term *test case* to refer to a finite sequence of user events $e_1 e_2 e_3 \dots e_n$ that are transmitted to the system under test. In the

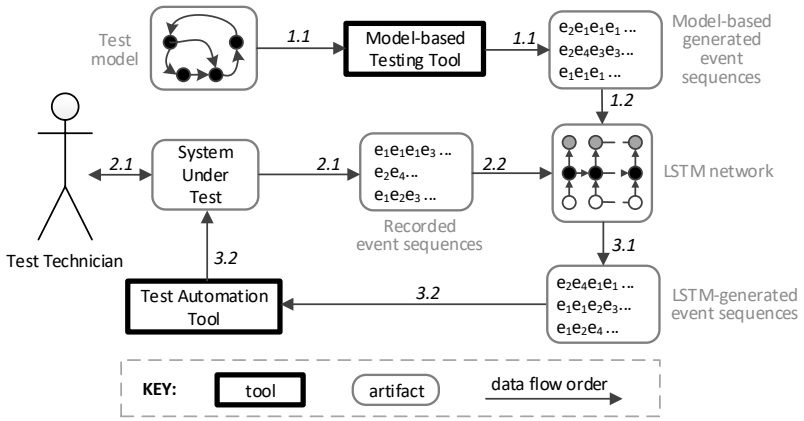
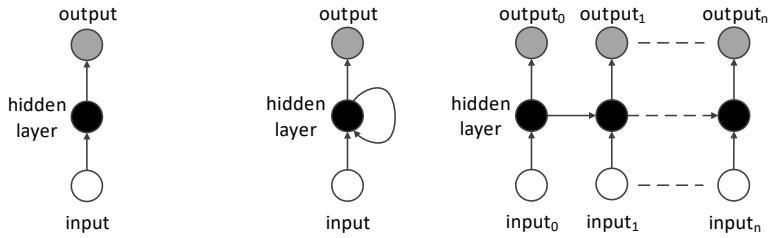


Fig. 1: The overall approach.

context of DTVs, a user event is a remote controller key press. A test case *fails* if the system under test crashes/halts during test case execution. Hence, our definition of faults include only crashes of the system under test. We defined a system crash as "being unresponsive for 5 seconds". Unresponsiveness of the system can be automatically detected, and the sequence of events that caused the unresponsiveness, hence crash, are logged for further inspection.

Our general approach is depicted in Figure 1. We compose two groups of test cases to train the neural network. The first group of test cases (step 1.1) are automatically generated from an existing high-level test model. (In principle, this group can be replaced with any test suite that is either manually or automatically created.) This group represents the general usage behavior of the system under test, and are used for functional testing of the system. The second group contains test cases recorded during exploratory testing activities (step 2.1); this set is used for robustness testing, and thus represents potentially error-prone usage behavior. We combine all the test cases from the two groups and feed into LSTM for training (steps 1.2 and 2.2). Then, we use the trained network to generate new test cases (step 3.1), which can be executed on the system under test (step 3.2).

Although the test cases from the exploratory testing sessions are expected to be more "valuable" in the sense that they are supposedly more liable to reveal faults, in our approach we do not solely use these test cases, but rather combine them with test cases generated from a model as well. This is because the size of the test cases recorded during exploratory testing is limited due to the human-centric nature of this activity. Neural networks, however, require a large amount of input for training; inputs that are too small lead to over-fitting (i.e. memorization) problems. We remedy this problem by populating



(a) A traditional neural network (b) RNN (c) RNN in unfolded form

Fig. 2: Traditional vs. Recurrent Neural Networks (RNNs)

the data set with test cases generated from a model to a size that is reliably large to be used for training the LSTM.

3 Background

In this section, we provide background information about long-short term memory networks and training these networks for sequence modeling.

3.1 Long Short-Term Memory Network

Long Short-Term Memory (LSTM) network [25] is an artificial neural network [23] that learns from the sequential input data. It is a type of *Recurrent Neural Network (RNN)*. As a difference from traditional feed-forward neural networks [23], RNNs keep contextual information in the form of an internal state. Figures 2(a) and 2(b) simply illustrate a traditional feed-forward neural network and RNN, respectively, where the network in the latter has a feedback loop from the hidden layer to itself. This feature makes RNNs very effective for learning and predicting order/sequence of events. Figure 2(c) depicts the RNN in Figure 2(b) in unfolded form. Neural networks employ custom modules that transfer state information from one module to the next. These modules can be considered as analogous to neurons in a human brain. They implement a neuron activation function — typically a step function, a non-linear function such as sigmoid [23], or more recently, rectified linear units [41]. When used in RNNs, these neurons also transfer activation states of previous time steps of a sequence. Although RNNs are effective for learning short-term dependencies, they fall short for learning dependencies among elements of a sequence that are far apart from each other. In other words, there is no guarantee for an RNN to learn correlations of fed data that are relatively distant in time.

LSTM network is a special type of RNN that is capable of learning both *short term* and *long term* dependencies among a sequence of inputs. This capability is achieved by employing a sophisticated transfer function that propagates state information [21]. This function is composed of multiple subcomponents, called gates. Each of these gates involves an independent vanilla

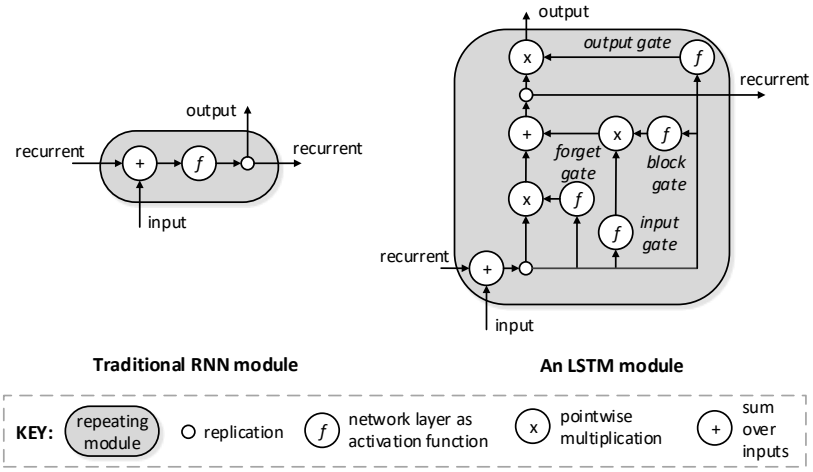


Fig. 3: Traditional RNNs vs. LSTM Networks

neural network of its own, and it controls the state by filtering, adding, or updating information during the state transfer. Altogether, based on the input training data, an LSTM network learns what to forget, what to remember, and for how long. In other words, the default behavior of an LSTM network is to remember a context until it is automatically decided that it is no longer of value, whereas a vanilla RNN has no default behavior for keeping the context. Hence, RNNs tend to lose context when the sequences get lengthy.

The difference between a traditional RNN module and a typical transfer module in LSTM is depicted in Figure 3. An RNN module combines the previous state information vector (i.e., recurrent) with the input vector, which is fed to an activation function. The output of the function is used both as the output and the state information to be propagated. On the other hand, an LSTM module employs 4 network layers as activation functions. These network layers are also trained along with the whole network for learning which part of the input should be passed through or filtered out (forget gate), which part should be added (input gate) to which part (block gate) of the current state, and which part of this state should be reflected to the output (output gate). There can be variations of an LSTM design [19,9,21] and there exist several practical applications mainly concentrated on language modeling [18]. The implementation of Google Translate [61] can be provided as an example for one of the most popular applications at the time of writing this paper.

3.2 Training RNNs for Sequence Modeling

Sequence modeling has been a major topic in natural language processing field. Natural languages have a predefined dictionary of words where words are generally formed by a sequence of characters. Since the sequence of characters for a specific word never changes, modeling the sequence can be simplified by tokenizing every word in the dictionary. First, words are considered as ordered in a predetermined fashion, usually the lexicographical order. Second, a representative vector is assigned for every different word according to its index in the ordered representation. If an ordered dictionary contains N words, each word is represented by an N dimensional vector where all the dimensions are set to zero but the dimension with the index corresponding to the word's order in the dictionary is set to 1. This vector is called a one-hot vector.

The major problem with one-hot representation is that the dimensionality can be extremely high in some scenarios. Considering the English language contains approximately 170k different words in its dictionary, one-hot representation is clearly an infeasible method. For high dimensional problems, word embeddings are used. Each word is, now, represented by a lower dimensional embedding instead of an N dimensional one-hot vector. Briefly, one-hot representation assigns a unique dimension per word whereas embedded representations use a lower dimensional hyper-plane on which words are located as distant as possible to other words.

In our case, we have a relatively small dictionary of events that we can tokenize in such a way that each token can even be represented by a single-byte character. Since we do not have high dimensional data, we choose to directly use one-hot representation on a character level RNN.

Training an RNN by using a back-propagation algorithm is tedious since RNNs are cyclic graphs. Having a cycle makes the standard procedures inapplicable to the training process. It turns out that we can always form a directed acyclic graph (DAG) from a cyclic directed graph by a procedure called *unfolding* through time. There is no need to store copies of weight matrices for different time steps as they are the same at every step in the sequence. This idea enables and facilitates the training by directly using the standard back-propagation methodology on the unfolded network. Note that, the input to the unfolded network becomes the input of the unfolded sequence all at once. This widely established method for training an RNN is called Back-propagation Through Time (BPTT) [55] as described in the following subsection.

3.3 Back-Propagation Through Time

The system under test (SUT) is supplied with a set of input events (volume-up, power-on, go-right, etc. for a DTV system). These events form a sequence. This sequence can be infinitely long. Looking at a cross-section of this sequence, the question is "can we learn to predict only the next event". For instance, by looking at the first k events, can we guess the $(k+1)^{th}$ event? Clearly,

having a small k will limit our prediction ability. Optimally, one would want to consider all the events coming before the last event in order to accurately predict the next one. Looking through a long history of past events would provide more information to learn from. Unfortunately, this comes with a trade-off. Being able to predict by using all the previous historical data is practically not tractable since it would require vast amounts of data storage and high computational power for training in a generic manner. To remedy this problem, we need to make a simplification. This simplification is to limit the size of the history of events considered for prediction of the next event.

Back propagation (BP) is a training algorithm widely established due to its efficiency. Unfortunately, RNNs are not suitable for using BP directly for training. In order to make an RNN suitable to be trained by BP, *Back-propagation Through Time (BPTT)* method is used. Fortunately, BPTT method makes the same simplification we described above. If the event history is limited to a window of the last k events, then the RNN can be unfolded k times through time and be converted to an equivalent standard neural network. An RNN normally would require only one event input per its time step, but the unfolded version of it requires a window of k events. Therefore, if we limit our attention to a window of the last k events, we are able to convert the RNN to an unfolded standard neural network on which the standard BP can be used for training. This whole procedure is called BPTT. During a BPTT training, we supply a window of k events plus the event that immediately follows this window, and the network learns this last event as a function of the events in the window. Therefore, supplying a window of events corresponds to supplying a single sample to the RNN. Since, we do not want to skip training samples, a legitimate way of training an unfolded network is to supply windows of events that are shifted by one event at a time.

The input for the unfolded neural network is a window of k events from the real sequence at any time step. The output is a prediction of the next event by the neural network. In reality, at test time, we randomly supply our RNN with a so-called burn-in sequence of k events for randomizing its initial behavior. We then use the last k events for predicting the next one. Once we have the next event, it is considered as a real event that just happened. We then slide our window of last k events to include the lastly predicted event and predict the next event. This can go on forever. This method is a widely established way of generating a sequence of events.

In the following section, we introduce our problem statement in the context of an industrial case study. Then, we describe our approach and the LSTM design we used for addressing this problem.

4 Industrial Case Study

We evaluate our work in the context of black-box testing of DTVs at Vestel. The company serves more than 900 customers covering ∼150 different brands in 157 countries. There are approximately 100 software engineers and 100 test

engineers/technicians employed in the R&D department. Vestel manufactures about 10 million DTVs annually. The code base is approximately 5 million LoC in total. There is a dedicated software test group in Vestel that performs various types of tests such as performance tests, certification tests, and functional tests. In this work, we focus on black-box robustness testing (named as "torture tests" in Vestel), where the goal is to detect crash failures only. There are hundreds of test suites being used for testing DTV systems. Some of them are manually created, while some others are automatically generated using model-based testing, where the models contain thousands of states and transitions. Black-box test execution is largely automated[2] by an in-house developed tool, namely Vestel test automation system (VesTA) [46,35]. This tool automatically sends remote controller key signals to the DTV by executing an input test script.

Recall from Figure 1 in Section 1 that we combine test cases from two sources. The first are the test cases recorded during exploratory tests. The second are those generated from a model of the system. MaTeLo[3] was being used by Vestel as the MBT tool at the time of conducting this study. There were already existing test models created with this tool, which conform to an Extended Markov Chain (EMC) formalism [22]. For this reason, we used this tool and formalism in our case study as well.

A Markov chain is basically a FSM, in which probabilities are defined for state transitions [56]. The system may change its state from the current state to another state, or remain in the same state, according to a probability distribution. The EMC formalism that is adopted by MaTeLo involves features of the state chart [24] and LTS [54] formalisms as well. First, they can be hierarchically composed just like state charts. Second, each transition can be associated with input and output labels just like LTS. In fact, transitions can be labeled with multiple inputs and outputs in an EMC model. Hereby, inputs and outputs are observable actions exchanged with the system. Each of the inputs is associated with a probability value such that the sum of these values for the alternative inputs is equal to the transition probability. Therefore, EMC specifies two levels of probabilities; *i)* the probability to select the transition and *ii)* the probability to select an input given that the transition occurs [22]. There was a single input defined for each transition in our case. This input includes the sequence of events (i.e., remote controller key presses) necessary to perform the corresponding transition. No output labels are specified since the created test models are used for detecting crash failures only. Test models that are created for functional tests also involve output labels, where the expected visual output is specified in the form of TV screen snapshot images [33].

The main focus for our test framework is to capture crashes, if any. The system does not suppose to crash even though the user can press any button of the remote controller in any order at any speed. Our test data is supplied by Vestel, a big company producing numerous different Digital-TV systems

[2] Some tasks require physical access to the system and cannot be automated.

[3] `http://www.all4tec.net/`

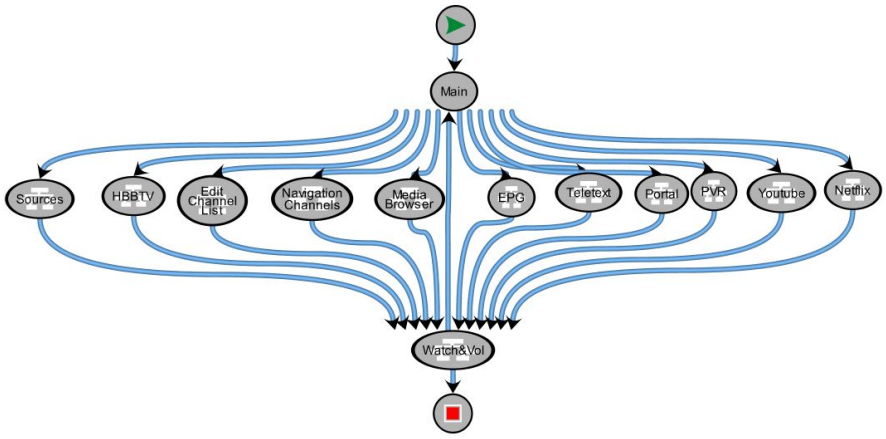


Fig. 4: The top-level test model.

and configurations. Code-base is relatively mature. So, it is challenging to find scenarios that trigger (new) system crashes. Yet, we obtained promising results by identifying several of those as explained later in Section 5.4.

In our case study, we used particularly the test model depicted in top-level view by MaTeLo as shown in Figure 4. This model is designed by Vestel test architects based on usage profiles that are collected from end users. It includes the most essential TV features that should be tested, such as the media browser, channel list editing, Youtube portal, channel navigation, etc. The model states that the DTV starts in the initial state denoted as "Main" in Figure 4, then transitions to one of the features. When in the state of a feature, we expect some number of feature-specific actions to take place, such as navigating up and down in the channel list, or selecting a Youtube video by navigating in the left/right/up/down directions. Once these actions finish, the DTV transitions to the "Watch&Vol" state, where volume adjusting actions take place. Finally, the DTV goes back to the initial state (denoted by a back edge from "Watch&Vol" to "Main").

Note that the model in Figure 4 is far simpler than what a complete model of a DTV would be. In fact, there already exist other sophisticated and detailed DTV feature models specified by the Vestel testing group. However, we deliberately assumed those sophisticated models do not exist and did not use them because such models are labor-intensive to specify and require mapping transitions and state-related actions to low-level remote controller key press event sequences. We wanted to experiment with an approach where we start from an under-specified model that can be defined by a test architect without much effort. Vestel testing group stated that for a test engineer familiar with the MaTeLo tool, preparing the model in Figure 4 including the mappings from states to remote controller key event sequences takes less than half a day.

Test cases are generated from the model using the MaTeLo tool based on the following flow:

1. At the Main state, randomly pick a TV feature (e.g. Media Browser). Emit the remote controller key press sequences to transfer from Main to that state.
2. Emit predefined remote controller key event sequences to utilize basic functionality and navigation in the entered state. These sequences are defined by the test architect when specifying the model.
3. Emit remote controller key event sequences to exit the state and enter the Watch&Vol state.
4. Emit remote controller key sequences to adjust volume.
5. Repeat this flow until a termination condition is met.

The sequences to emit in Step 2 above involve randomization to increase variation. For instance, for channel navigation, N number of "down" key presses are emitted followed by M number of "up" key press events, where N and M are randomly picked integers in a range pre-specified by the test architect. Potentially, an infinite amount of sequences can be generated. Therefore, we need to check in Step 5 for whether the process should be terminated. For this, one the following three criteria can be used: (1) reaching a tester-specified threshold for the number of event sequences generated, (2) reaching a tester-specified time-limit, (3) reaching a tester-specified ratio of coverage of the edges in the model. For our experimental evaluation in Section 5, we used the third criterion with full coverage of the edges.

The generated test cases are output as executable Python code that can be run using Vestel's test automation tool, VesTA. A snippet that shows a possible output to test the Media Browser feature is below:

```
sendKey("media", pause=5)
sendKey("ok", pause=2)
sendKey("right", pause=1)
sendKey("down*22", pause=2)
sendKey("up", pause=2)
sendKey("up*22")
sendKey("ok", pause=5)
sendKey("right*2", pause=2)
sendKey("left*2", pause=2)
sendKey("exit*3")
sendKey("back", pause=10)
# Watch&Vol state
sendKey("voldown*14", pause=2)
sendKey("volup*47", pause=2)
sendKey("voldown*22", pause=2)
```

Here, the remote controller key to be pressed is specified as a string. For instance, `"ok"` refers to pressing the OK button, `"right"` is the right arrow key, etc. When executing the script, VesTA literally sends the specified remote controller key press events to the DTV under test. If a `pause` argument is provided, VesTA waits for the given amount of seconds before executing the next

statement. A star character (`*`) denotes repetition of the preceding event; for instance, `"down*22"` instructs VesTA to send the down arrow event consecutively for 22 times. Repetition factors are randomly picked according to the specification of the test architect; another generated instance of the sequence above may include `"down*17"` instead of `"down*22"`. This way, MaTeLo generates hundreds of remote controller key press event sequences that are similar but not exactly the same.

During exploratory testing, the test technician holds the remote controller in her/his hand, and presses the keys. These key events are not only transmitted to the DTV under test, but also captured and recorded by VesTA in the same Python format shown above so that the testing session can be re-executed in the future if needed.

We shall note that the model-based and exploratory testing practices explained in this section are routine procedures at Vestel; they have not been developed or altered specifically for us. The novelty of our approach is to combine these existing practices using machine learning techniques with the goal of enhancing their effectiveness and reducing the human-labor costs.

5 Evaluation

We define the following **research questions** to evaluate our work:

RQ1: *Can we detect new faults by executing test cases generated by LSTM?*

RQ2: *How effective and efficient are test cases generated by LSTM compared to those that are randomly generated?*

RQ3: *How the effectiveness and efficiency of test cases generated by LSTM change based on whether or not test cases recorded during exploratory testing activities are used for training LSTM?*

The first question is concerned with the usefullness of the approach. We would like to know if we can detect faults that could not be detected with the existing test cases or during previous exploratory testing activities. The second question is to make sure that faults are not detected by chance. Hence, we would like to compare our approach with random testing as the baseline. The comparison criteria is *effectiveness* and *efficiency*. We measure *effectiveness* in terms of the number of new faults detected. We measure *efficiency* in terms of the number of unique faults detected per unit of time. The last research question is regarding the impact of test cases recorded during exploratory testing activities as input for training. We could have trained LSTM by just using test cases automatically generated from a test model. We would like to know how much we gain with respect to test effectiveness and efficiency, if we do at all, when we utilize recorded exploratory testing activities for training LSTM.

We evaluated our approach in the light of these research questions. In the following, we provide the details of our experimental setup and discuss the results obtained.

5.1 Data Collection and Experimental Setup

We compiled two sets of test cases as our data to be used for training and evaluating the LSTM. Our first set comprises 221,801 remote controller key press events (plus a total of 463 minutes of pause directives) generated from the system model as discussed in the previous section. We used full edge coverage of the model as the termination condition of the generation process. Our second data set comprises test cases recorded during the most recent exploratory testing sessions at the time of writing this paper. The test technicians were unaware of the work we intended to pursue, and they conducted their usual exploratory testing procedures. We collected a set of test cases comprising 2,802 key press events (plus a total of 237 minutes of pause directives).

To be able to answer our research questions, we performed a comparison among the following methods:

- *AR (All Random)*: A test case produced by randomly picking key press events from the set of all possible key press events. There are 202 events in this set.
- *SR-MBT (Selected Random based on MBT)*: A test case composed of a completely random sequence of key press events using only the keys that occur in the test cases generated with MBT. There are 37 such key events.
- *SR-ET (Selected Random based on ET)*: A test case composed of a completely random sequence of key press events using only the keys that occur in the recorded ET sessions. There are 32 such key events.
- *WR-MBT (Weighted Random based on MBT)*: A test case composed of a random sequence of key press events, where the probability of occurrence for a key press event is proportional to the frequency of occurrence of that event in the test cases generated with MBT.
- *WR-ET (Weighted Random based on ET)*: A test case composed of a random sequence of key press events, where the probability of occurrence for a key press event is proportional to the frequency of occurrence of that event among the events recorded during ET.
- *MBT*: Test cases generated with MBT.
- *ET*: Manually performed ET activities.
- *LSTM-MBT (LSTM trained based on MBT)*: Test cases produced by an LSTM network that have been trained with the test cases generated with MBT.
- *LSTM-ET (LSTM trained based on MBT and ET)*: Test cases produced by an LSTM network that have been trained with both the test cases generated with MBT and those recorded during ET.

In the random methods, we used a uniformly-distributed pseudo-random number generator. Also in those methods, each key press events is followed by a 1 second pause event to prevent the event buffer of the DTV from becoming full and eventually dropping events or becoming unresponsive. Note that MBT test cases include pause events, too, as specified by the test architect. ET test

cases naturally include pause events between key press events, because they are conducted by a human.

5.2 Data Normalization, Encoding, and Decoding

Before feeding the test cases to LSTM as training data, we normalize them by unfolding all the event repetitions and translating the pause specifications to pseudo events. For example,

```
sendKey("up*3", pause=2)
sendKey("right*2")
```

is converted to

```
sendKey("up")
sendKey("up")
sendKey("up")
sendKey("WAIT-1-SEC")
sendKey("WAIT-1-SEC")
sendKey("right")
sendKey("right")
```

We then encode the normalized test case by converting each event (hereby, each line) to a unique 1-byte character. Because there are fewer than 256 different remote controller key types, this is a straightforward one-to-one translation. The encoded data are used for training LSTM. Hence, the output of LSTM (i.e. the generated test cases) is in encoded format as well. We pass this output through a decoding phase to convert the 1-byte character encodings back into human-readable and executable Python source. Decoding is simply the reverse of the one-to-one mapping applied during encoding.

5.3 LSTM Network Setup

Test performance of neural networks are known to be sensitive to their hyper-parameters and initial values of the training parameters. Training of neural networks needs numerous parameters to be fine-tuned for generic test performance under unforeseen scenarios. These parameters include the number of hidden layers, number of neurons in each hidden layer, learning rate, number of unfoldings (for recurrent neural networks), batch size, etc. Type of the neurons and the training algorithm are also important hyper-parameters.

In this work, we used the *karpathy/char-rnn* open source project [31]. Our training algorithm is Adam optimizer [32]. Despite using more memory, Adam optimizer is known to be less sensitive to initialization of neural network parameters. This is a must-have property for optimizing a sensitive cost function containing numerous parameters. Adam optimizer is a variant of the gradient descent (GD) method. GD requires all the training samples to be considered

per one epoch of training. This is normally not feasible as there are normally too many training samples. Although GD is accurate, it takes too much time to converge. Depending on the training scenario, and available computational power, convergence time can reach up to months, or even years. A simplification of this approach is called Stochastic Gradient Descent (SGD), where training epochs are divided into iterations of isolated training of each training sample. SGD is known to converge extremely fast compared to GD, however, it is also considered unstable around local extrema of the optimized cost function [8]. Moreover, considering the parallel pipeline of today's CPUs and GPUs, processing only one of the training samples per iteration causes under-utilization of parallel computational power. A compromise between GD and SGD is called mini-batch training in which a finite number of sample batches are populated from the training dataset per iteration [10]. Mini-batch training utilizes the parallel pipeline of CPUs and GPUs; moreover, it is considerably more stable than SGD. Adam Optimizer is a variant of SGD that is less sensitive to initial hyper-parameter values.

Neural networks, in general, need the training data to be sufficiently large to avoid the memorization/overfitting problem. To make sure that our LSTM network does not have this problem with our data, we applied the following validation process: We randomly picked 90% of the data set to be our training set. Remaining 10% is divided equally to be used as validation and test sets. Our total sample size is 266,598 key press events, including the pseudo "WAIT-1-SEC" events. Hence, the training set contained approximately 240k events, whereas the validation and test sets included nearly 13k events each. We used Adam optimizer to train on the training portion of the dataset. We continued training until the validation set performance did not increase anymore. Finally, we cross-checked our validation performance with the test performance, and found that they are similar. This means that LSTM training did not memorize the data; instead, it learned a generic summary of the training dataset. Generic training enables us to generate completely new test cases resembling the input data.

Next, we experimented with various parameter settings, and found the following setup to give the best results:

- We used 30 steps of unfolding through time. In other words, each of our training samples is a window of 30 consecutive events from the whole test sequence. So, the LSTM model considers 30 events for predicting the next event. Our input data contains 48 unique key press events. The window is shifted by 1 for each sample. Increasing the window size (unfolding steps) too much, increases the training time. On the contrary, choosing this parameter too low severely restricts the generative power of the LSTM. The unfolding value that we used is a generous amount, and supplies the LSTM network with the correlations of characters that are farther away from each other, which is where LSTM networks are notoriously successful on.
- A batch size of 4000 samples has been used as the mini-batch size out of approximately 240k samples, meaning that each epoch of our training

contained 60 iterations. This increased the utilization of parallelization available in our CPUs and GPUs while at the same time induced enough stability of training convergence. Learning rate of the Adam optimizer is set to 0.002, which is the default value in the library we used.
- After experimenting with various hyper-parameters, we used two hidden layers of LSTM neurons and 128 neurons per hidden layer. Once these hyper-parameters are set, the constructed LSTM turns out to have 228,400 different parameters (i.e., weights and biases of neurons) to optimize.
- We trained the LSTM network for 10000 epochs, and saved the network parameters that gave the best error rate on the validation data set.

As the last step of our approach, the trained LSTM network is used for generating event sequences [20]. To be compatible with our training, for the first 30 unfolding time steps, we supply a random sequence to the network for initializing its internal hidden state vector. The network then predicts a likelihood of the next character in the sequence. We sample a new character from this likelihood. The sampled character is then emitted, and fed to the network as a new input. This changes the internal state. We keep this procedure going until we sample a predefined number of new characters. Finally, the generated sequence of characters is decoded by mapping each character to its corresponding remote controller key press event.

5.4 Results

In this section we present the results obtained by the testing methods. In total, 12 unique faults were detected during all the test runs. Some of these faults were revealed multiple times. Each of the 12 faults lead to distinct failures. This information has been verified by the Vestel testing group. Table 1 presents the overall results. Hereby, the first column lists all the compared approaches. The following 12 columns indicate which of the detected 12 unique faults are revealed by each of the approaches. The last 3 columns list the *total number of unique faults revealed*, *total duration* and *efficiency* of each approach. Efficiency is basically calculated as the number of revealed faults divided by total duration.

LSTM-ET approach revealed 6 unique faults (F1–F6), of which 4 have *not* been detected by any other testing method (F1, F2, F5, F6). Among these, F1 and F5 are particularly important, because these two have been classified by the Vestel testing group as "critical" — the fault type that needs immediate attention — and have been entered into the company's bug-tracking system. F1 is a *crash* problem where the TV does not respond to any commands; F5 is a *TV reset* problem that forces the TV to reboot itself. Other faults are a *no video* problem on the TV channel (F2), a *wrong source switching* problem (F3), *black screen* fault (F4), and *wrong audio* at the Netflix screen (F6). LSTM-MBT revealed 5 unique faults. Among these, F9 was not detected by any other method; and F3 was not detected by a non-LSTM method. LSTM-MBT and LSTM-ET together revealed 9 unique faults.

Table 1: Detected unique faults with the test cases obtained via the compared methods.

Method	Unique Faults Detected												# of Faults Revealed	Duration (hrs)	Efficiency
	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12			
AR													0	50	0.00
SR-MBT												✓	1	50	0.02
SR-ET				✓									1	50	0.02
WR-MBT				✓									1	50	0.02
WR-ET				✓									1	50	0.02
MBT				✓			✓	✓					3	21	0.14
ET										✓	✓		2	4	0.50
LSTM-MBT			✓	✓			✓	✓	✓				5	15	0.33
LSTM-ET	✓	✓	✓	✓	✓	✓							6	15	0.40

The ET method detected only 2 faults, but these faults were not revealed by any other method. This shows once again why ET is an indispensable testing activity despite the manual effort involved. The two faults revealed by ET, F10 and F11, are about manual channel searching and related to channel frequencies. They require entering particular frequency values as search criteria. In that sense, they require detailed domain knowledge, and hence were detected only by the human testers.

So, returning back to our research questions, we see that

RQ1: We were able to detect new faults with our LSTM-based approach. In fact, half of the all the detected faults (F1, F2, F3, F5, F6, F9) are only detected with LSTM-based approaches.

RQ2: Our approach outperforms all the baseline approaches that are based on random testing. It also significantly outperforms model based testing although test cases that are generated from the test model are used for training the LSTM network. We see that ET is the best in terms of test efficiency; however, this is the only approach that is manually applied and as such incurs a high cost.

RQ3: The number of faults detected by LSTM-MBT and LSTM-ET were 5 and 6, respectively within the same testing duration (15 hrs). An important observation here is that these sets of faults are mainly complementary, i.e., the majority of the faults detected with LSTM-ET are not detected with the other approaches. We can see in Table 1 that only two of the revealed 6 faults are common between LSTM-MBT and LSTM-ET. LSTM-ET revealed 4 faults that are not detected with any of the other methods.

5.5 Threats to Validity

The number of detected faults with all the evaluated methods is 12 in total. This hinders conclusion validity. The main reason for detecting a low number of faults is that we focus on crash failures only. Unfortunately, we could not mitigate this threat by performing controlled experiments with seeded faults. We conducted an industrial case study instead, where the level of control is inherently low. Even the test engineers we worked with perform only black-box testing and as such, they do not have access to source code.

Our evaluation is subject to external validity threats [57] since it is based on a single case study. More case studies can be conducted for generalizability of results. We did not foresee any internal validity threats because we neither selected human subjects as participants nor we performed any instrumentation for measurement. Our measurements just involve external observations and offline analysis of real products. Test technicians merely performed their regular tasks without any external interference, while their actions being recorded.

Our LSTM setup and configurations can lead to construct and external validity threats. However, we documented all the implementation details regarding the utilized libraries and parameters for facilitating the repeatability of our experiments. We also followed a rigorous validation process to avoid memorization and overfitting problems for the LSTM network. Nevertheless, there exist an inherent randomness in the proposed test case generation method. To investigate the affect of randomness, we devised 5 methods that employ random testing. All these methods detected at most one fault in 50 hours of testing. The other approaches including ET, MBT and LSTM-MBT/ET were applied for 4, 21 and 15 hours, detecting 2, 3, 5/6 faults, respectively. Although the total number of faults is low, we observed increased efficiency with respect to pure random methods, considering the time invested in testing. In the following, we further discuss how a quality training can be employed to mimic the behavior of human test technicians.

5.6 Effective Training for Mimicking the Behavior of Human Test Technicians

We used two source of inputs for constructing our training data sets. Firstly, we employed a simplified model-based representation of the system under test (SUT). This part can generate sequence of events that traverse the most essential states of the SUT. Secondly, we have test technicians and the sequence of events generated by them. This part encapsulates the intrinsic knowledge of the test technicians for making the SUT crash. Our major aim in this study is to be able to capture the internal know-how of the human testers, and create an expert system that can mimic their behavior for many more generated scenarios. While doing this, we do not want to memorize the exact behavior of the test technicians, as this will never create new event sequences. We already have those sequences recorded anyway. Instead, what we aim is to make the learned model in such a manner that it can generate new events in a so

called "creative" manner. In other words, we want our machine learning (ML) algorithm to learn in a generic fashion, and to capture the essence of the event sequences that are highly probable to cause system crashes. This can be visualized as creating synthetic human test technicians who behave similar to their real counterparts, but who still act uniquely on their own.

Our main goal is to capture the behavior of human test engineers that lead to unseen crashes of the SUT. Human test engineers, due to their domain knowledge, have a better insight regarding what will break the SUT. Unfortunately, access to human test engineers is highly limited. For our scenario, we had extremely limited amount of human test data available at hand. We also believe that this is a frequent scenario for a wide range of companies. Unfortunately, machine learning methods, especially RNNs, demand big data to be trained with high quality. As a result, our only option was to feed and augment the training set with data that is sampled from a correct model of the SUT. In order to clarify the effectiveness of the proposed method, we designed our experimental setup considering 9 different scenarios (AR, SR-MBT, SR-ET, WR-MBT, WR-ET, MBT, ET, LSTM-MBT, LSTM-ET) whose details are available in Section 5.1. For a specific execution of ML training such as LSTM-MBT and LSTM-ET, we systematically divided the underlying data set into training (240k samples), validation (13k samples) and test (13k samples) partitions. We pushed the limits with a 90%/5%/5% partitioning in order to put as many samples as possible to the training partition, since we do not have enough data, and at the same time we want to minimally dilute our data with synthetic samples. We trained our model using the training partition. We checked our performance on the validation partition until the training is finalized. We picked the best iteration to halt by maximizing our validation performance. However, this is still a biased performance since the validation set is used for deciding where to stop the training. The real performance is the performance on the test data set, which is never seen by the trained model. We reported the test performances. Due to the insufficient human test engineer data, and the need of supplying large data to ML training, we augmented our data as much as possible. We used the maximum possible amount of synthetic data that didn't inhibit the quality of LSTM training. We also used the maximum amount of training data that also satisfied this scenario (90% of all the data available).

In ML, there are two frequently used keywords for defining the quality of learning. One is called *over-fitting*, the other is called *under-fitting*. An over-fitting expert system is said to have "memorized" the data such that it loses its creativity and always produces the same results when presented with a specific state. An over-fitting expert would always detect the errors in the test sets if it has encountered them during its training. But, this would definitely not be beneficial, since we are trying to capture the essence of test technicians' behavior, not what they exactly did. On the other hand, an ML expert system that is said to be under-fitting is an expert system that needs more training for being effective. An under-fitting expert system is not expected to produce quality outputs. A sweet-spot occurs just between over-fitting and

under-fitting, where the system is considered to have learned in a good manner, and can produce generic creative outputs by itself. This means, it has sufficiently learned, yet not memorized its training data sets. In order for this to happen, a well trained system can and is allowed to make mistakes on its own training sets, because this enhances their performance on never-seen inputs. Such a system is considered to have gone a qualitative ML training phase.

Briefly, our trained ML expert system is complementary to the model-based synthetic event generator. They are designed to be used together. A good quality training can cause the trained ML expert system to make mistakes on the training set. A real threat to validity would occur if the trained system was unable to generate event sequences that cause system crashes. Yet, in our approach, the trained system generates crash sequences that were seen before, and moreover, it also generates sequences that causes completely new types of crashes that had never been encountered before.

6 Related Work

Model-based testing has been studied for decades. There exist numerous techniques [42] and tools [11] proposed for this approach. These techniques and tools employ test models that are expressed in various formalisms such as finite state machines [48], event sequence graphs [5,6], event flow graphs [38], and Markov chains [56]. In our approach, the test model is implicitly represented by an LSTM network.

One of the drawbacks of model-based testing is manual creation of models, which is an effort-consuming and error-prone process [16]. Therefore, several approaches have been proposed to automatically synthesize test models or automatically refine/extend existing models. For example, LBTest tool [36] automatically creates a test model in the form of a finite automaton based on a set of inputs executed on the system under test and observed outputs. Observe-Model-Exercise* [43] uses a possibly incomplete model of the graphical user interface of an application, which is expressed in the form of an event flow graph. This model is iteratively extended with new events that are detected during the execution of the application. A variant of this approach [3] is applied for mobile applications as well. Other similar techniques include capture-replay testing [13] and automated feedback-based techniques that aim at increasing either code coverage [15,40] or system behavior coverage [60,45]. None of these approaches exploit knowledge and insight of experienced test engineers. The model we used to populate the training data for LSTM is a non-sophisticated model that can be specified by a test engineer without going through a labor intensive process.

Mesbah et al. [39] automatically construct a test model for Web applications in the form of a state machine. The model is constructed by automatically crawling the application. Testilizer [14] extends this approach by augmenting information derived from human-written test suites. In our approach, we exploit recorded actions during exploratory testing activities. These records can

be replaced with an existing test suite as well. However, we only exploit these records as external events for black-box testing without any access to the implementation of the system. This is not the case with Testilizer, which makes use of the DOM model of the tested Web application [14]. As a result an explicit model of the application is generated in the form of a state machine. In our approach, the model representation is implicit in the form of an artificial neural network.

Deterministic approaches like GUI ripping [37] can be employed to automatically infer a model of the system. However, the related techniques and tools have limited applicability since they do not work in a purely black-box fashion. They run on the same machine as the system under test, assuming that GUI components (e.g., buttons, labels) [44] or a document object model (e.g., as in HTML) for Web applications [39] are available. These elements are used for identifying states and state changes in the application. However, such elements might not be accessible for all types of systems and test setups. For instance, testers do not usually have any access to the internal events during the testing of embedded systems such as Digital TVs. They do not have any access to the GUI components either. Therefore, one has to infer state information and state changes only based on a sequence of external user events.

In general, there have been various approaches evaluated for automatically generating tests for Web applications [12]. These approaches are categorized into 3 types [12]: *i)* purely white-box approaches that create a model based on objects and their interactions, *ii)* purely black-box approaches that only make use of user-session data, and *iii)* hybrid approaches. Our approach is purely black-box. However, it is different from all the other black-box approaches in that an implicit model of the application is learned in the form of an artificial neural network. Other black-box approaches either directly replay recorded sessions [52], generate new test cases by combining existing ones and/or by modifying them [12]. They do not learn a usage model.

Learning-based testing [36] has been introduced as a paradigm for black-box testing of reactive systems. Hereby, the goal is to iteratively construct a model of the system in the form of a state machine by using previously executed tests. To begin practical testing with this approach, a formal requirement specification is needed in the form of linear temporal logic. This formula is model checked on the constructed model to explore the relevant paths. These paths are used for test case generation. The test execution is used for inferring a refined model. Our approach does not require any requirement specification or another formal specification of any kind. We only exploit recorded tests to refine the existing artificial network model and generate test cases.

Exploratory modelling [49] was introduced for refining test models based on recorded activities of test engineers. Hereby, a test model is developed in the form of a state diagram. Then this model is manually updated based on observed states and behaviors during exploratory testing [53, 2, 27, 26]. We have previously proposed an approach [17, 16] for performing these updates automatically. In that approach we used Markov chains as the formalism to

express the test model. Execution traces that are collected during exploratory testing process are used for detecting missing parts in this model. The main challenge here is the mapping of recorded activities in the form of low level events to abstract states of the test model. One has to manually prepare a mapping specification for this purpose. In this work, recorded event sequences are directly supplied to an LSTM network, which learns the behavior that is implicitly represented in the network. The same network is then used for test case generation. Therefore, there is no need to provide a mapping of low level events or user actions to abstract states of a test model.

In general, machine learning techniques have been mainly applied for test oracle automation [4]. On the other hand, artificial neural networks have been previously used for supporting fault localization [59,58]. However, to the best of our knowledge they, in particular LSTM networks, have not been used for learning expected input sequences and thereby generating test cases. Relatively more primitive types of artificial neural networks have been used for automated test pattern generation for digital circuits [28].

7 Conclusion and Future Work

We introduced a novel application of LSTM networks for learning usage behavior of systems and automatically generating test cases. We applied our approach in the context of black-box testing of Digital TVs, where MBT and ET are being applied regularly as standard testing practices. As the training input to the LSTM network, we used both the test cases produced from a high-level model of the system for functional testing, and the recorded robustness test inputs of experienced test technicians. We proposed a complementary approach where the test cases generated by the trained LSTM network improve the number and variety of detected faults. The results show that MBT+ET+LSTM as a package achieves a higher fault coverage in a shorter time when compared to all the random methods tested.

The overall goal of machine learning is to automatically extract an already learned model by the human brain in the form of a mathematical model. Unfortunately, this extraction process demands big data. This is especially the case for RNN architectures. RNNs are accepted as the prominent choice for learning from sequence based data like the one we use in our case study. LSTMs, which are variants of RNNs, represent the state-of-the-art for sequential knowledge extraction. This was the reason behind our choice of this particular ML technique. However, there are many other ML techniques that can be employed for learning usage behavior and automatically generating test inputs. The appropriate choice would highly depend on the application context, the type of system under test, and the types of inputs consumed by this system.

Acknowledgment

We would like to thank the software developers, test engineers and technicians at Vestel Electronics for sharing their resources with us and supporting our case study. We also thank the anonymous reviewers for their comments on this paper.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, New York, NY, USA (2007)
2. Agruss, C., Johnson, B.: Ad hoc software testing: A perspective on exploration and improvisation. In: Florida Institute of Technology, pp. 68–69 (2000)
3. Amalfitano, D., Fasolino, A., Tramontana, P., Ta, B., Memon, A.: MobiGUITAR: automated model-based testing of mobile apps. IEEE Software **32**(5), 53–59 (2015)
4. Barr, E., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507 – 525 (2015)
5. Belli, F.: Finite state testing and analysis of graphical user interfaces. In: Proceedings of 12th International Symposium on Software Reliability Engineering, pp. 34–43 (2001)
6. Belli, F., Budnik, C., White, L.: Event-based modelling, analysis and testing of user interactions: approach and case study. Software Testing, Verification and Reliability **16**(1), 3–32 (2006)
7. Berner, S., Weber, R., Keller, R.K.: Observations and lessons learned from automated testing. In: Proceedings of the 27th International Conference on Software Engineering, pp. 571–579 (2005)
8. Bottou, L.: Stochastic gradient descent tricks. In: Neural networks: Tricks of the trade, pp. 421–436. Springer (2012)
9. Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, pp. 1724–1734 (2014)
10. Cotter, A., Shamir, O., Srebro, N., Sridharan, K.: Better mini-batch algorithms via accelerated gradient methods. In: Advances in neural information processing systems, pp. 1647–1655 (2011)
11. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-based testing in practice. In: Proceedings of the International Conference on Software Engineering, pp. 285–294 (1999)
12. Elbaum, S., Rothermel, G., Karre, S., II, M.F.: Leveraging user-session data to support web application testing. IEEE Transactions on Software Engineering **31**(3), 187–202 (2005)
13. Entin, V., Winder, M., Zhang, B., Christmann, S.: Combining model-based and capture-replay testing techniques of graphical user interfaces: An industrial approach. In: Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops, pp. 572–577 (2011)
14. Fard, A., Mirzaaghaei, M., Mesbah, A.: Leveraging existing tests in automated test generation for web applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 67–78 (2014)
15. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology **5**(1), 63–86 (1996)
16. Gebizli, C., Sozer, H.: Automated refinement of models for model-based testing using exploratory testing. Software Quality Journal (2016). Published online, DOI: 10.1007/s11219-016-9338-2
17. Gebizli, C.S., Sozer, H.: Improving models for model-based testing based on exploratory testing. In: Proceedings of the 6th IEEE Workshop on Software Test Automation, pp. 656–661 (2014). (COMPSAC Companion)

18. Gers, F., Schmidhuber, E.: LSTM recurrent networks learn simple context-free and context-sensitive languages. IEEE Transactions on Neural Networks **12**(6), 1333–1340 (2001)
19. Gers, F., Schmidhuber, J.: Recurrent nets that time and count. In: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, pp. 189–194 (2000)
20. Graves, A.: Generating sequences with recurrent neural networks. CoRR **abs/1308.0850** (2013). URL `http://arxiv.org/abs/1308.0850`
21. Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J.: LSTM: A search space odyssey. IEEE Transactions on Neural Networks and Learning Systems **28**(10), 2222–2232 (2017)
22. Guen, H.L., Marie, R., Thelin, T.: Reliability estimation for statistical usage testing using markov chains. In: Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 54–65 (2004)
23. Hagan, M., Demuth, H., Beale, M.: Neural Network Design. PWS Publishing, New York, NY, USA (1995)
24. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987)
25. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computing **9**(8), 1735–1780 (1997)
26. Itkonen, J.: Empirical studies on exploratory software testing. Ph.D. thesis, Aalto University (2011)
27. Itkonen, J., Mantyla, M.V., Lassenius, C.: Defect detection efficiency: Test case based vs. exploratory testing. In: First International Symposium on Empirical Software Engineering and Measurement, pp. 61–70. IEEE Computer Society (2007)
28. J. Štefanovič: A neural network algorithm for digital circuits test generation. In: Proceedings of the European Symposium on The State of the Art in Computational Intelligence, pp. 56–60. Physica-Verlag HD, Heidelberg (2000)
29. J.Bach: Exploratory testing explained. Tech. rep. (2003). URL `http://www.satisfice.com/articles/et-article.pdf`
30. Kaner, C.: Exploratory testing. In: Quality Assurance Institute Worldwide Annual Software Testing Conference (2006)
31. Karpathy, A.: char-rnn. `https://github.com/karpathy/char-rnn` (2015)
32. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
33. Kirac, M., Aktemur, B., Sozer, H.: VISOR: A fast image processing pipeline with scaling and translation invariance for test oracle automation of visual output systems. Journal of Systems and Software **136**, 266 – 277 (2018)
34. Lukac, Z., Zlokolica, V., Mlikota, B., Radonjic, M., Velikic, I.: A testing methodology and system for functional verification of general HbbTV device. In: Proceedings of the IEEE International Conference on Consumer Electronics, pp. 325–326 (2012)
35. Marijan, D., Zlokolica, V., Teslic, N., Pekovic, V., Tekcan, T.: Automatic functional tv set failure detection system. IEEE Transactions on Consumer Electronics **56**(1), 125–133 (2010). DOI 10.1109/TCE.2010.5439135
36. Meinke, K., Sindhu, M.A.: LBTest: A learning-based testing tool for reactive systems. In: Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation, pp. 447–454 (2013)
37. Memon, A., Banerjee, I., Nguyen, B.N., Robbins, B.: The first decade of GUI ripping: Extensions, applications, and broader impacts. In: Proceedings of the 20th Working Conference on Reverse Engineering, pp. 11–20 (2013)
38. Memon, A., Soffa, M., Pollack, M.: Coverage criteria for GUI testing. ACM SIGSOFT Software Engineering Notes **26**(5), 256–267 (2001)
39. Mesbah, A., van Deursen, A., Roest, D.: Invariant-based automatic testing of modern web applications. IEEE Transactions on Software Engineering **38**(1), 35–53 (2012)
40. Michael, C., McGraw, G., Schatz, M.: Generating software test data by evolution. IEEE Transactions on Software Engineering **27**(12), 1085–1110 (2001)
41. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th international conference on machine learning (ICML-10), pp. 807–814 (2010)

42. Neto, A.C.D., R.Subramanyan, M.Vieira, Travassos, G.H.: A survey on model-based testing approaches: A systematic review. In: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies, pp. 31–36 (2007)
43. Nguyen, B., Memon, A.: An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces. IEEE Transactions on Software Engineering **40**(3), 216–234 (2014)
44. Nguyen, B., Robbins, B., Banerjee, I., Memon, A.: GUITAR: an innovative tool for automated testing of gui-driven software. Automated Software Engineering **21**(1), 65–105 (2014)
45. Pacheco, C., Lahiri, S., Ernst, M., Ball, T.: Feedbackdirected random test generation. In: Proceedings of the 29th International Conference on Software Engineering, pp. 396–405 (2006)
46. Pekovi, V., Tesli, N., Resetar, I., Tekcan, T.: Test management and test execution system for automated verification of digital television systems. In: IEEE International Symposium on Consumer Electronics (ISCE 2010), pp. 1–6 (2010). DOI 10.1109/ISCE.2010.5523721
47. Rafi, D., Moses, K., Petersen, K., Mäntylä, M.: Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: Proceedings of the 7th International Workshop on Automation of Software Test, pp. 36–42 (2012)
48. Robinson, H.: Finite state model-based testing on a shoestring. In: Proceedings of the Software Testing and Analysis and Review West Conference (1999)
49. Robinson, H.: Intelligent test automation a model-based method for generating tests from a description of an applications behavior. Software Testing and Quality Engineering Magazine pp. 24–32 (2000)
50. Sak, H., Senior, A., Beaufays, F.: Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Proceedings of the 15th Annual Conference of the International Speech Communication Association, pp. 338–342 (2014)
51. Sivaraman, G., Csar, P., Vuorimaa, P.: System software for digital television applications. In: IEEE International Conference on Multimedia and Expo, pp. 784–787 (2001)
52. Sprenkle, A., Gibson, E., Sampath, S., Pollock, L.: Automated replay and failure detection for web applications. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 253–262 (2005)
53. Tinkham, A., Kaner, C.: Exploring exploratory testing. In: Proceedings of the Software Testing and Analysis and Review East Conference (2003)
54. Tretmans, J.: Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures, chap. Model-Based Testing and Some Steps towards Test-Based Modelling, pp. 297–326. Springer Berlin Heidelberg (2011)
55. Werbos, P.J.: Backpropagation through time: what it does and how to do it. Proceedings of the IEEE **78**(10), 1550–1560 (1990)
56. Whittaker, J., Thomason, M.: A markov chain model for statistical software testing. IEEE Transactions on Software Engineering **20**(10), 812–824 (1994)
57. Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A.: Experimentation in Software Engineering. Springer-Verlag, Berlin, Heidelberg (2012)
58. Wong, W., Debroy, V., Golden, R., Xu, X., Thuraisingham, B.: Effective software fault localization using an RBF neural network. IEEE Transactions on Reliability **61**(1), 149–169 (2012)
59. Wong, W., Qi, Y.: Bp neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering **19**(4), 573–597 (2009)
60. Xie, T., Notkin, D.: Tool-assisted unit-test generation and selection based on operational abstractions. Automated Software Engineering **13**(3), 345–371 (2006)
61. Yonghui Wu et al.: Google's neural machine translation system: Bridging the gap between human and machine translation. CoRR **abs/1609.08144** (2016)