11-2018

DSM: A specification mining tool using recurrent neural network based language model

Tien-Duy B. LE
Singapore Management University, btdle@smu.edu.sg

Lingfeng BAO
Zhejiang University

David LO
Singapore Management University, davidlo@smu.edu.sg

DSM: A Specification Mining Tool using Recurrent Neural Network Based Language Model

Tien-Duy B. Le
Singapore Management University
Singapore

Lingfeng Bao*
Zhejiang University City College
China

David Lo
Singapore Management University
Singapore

ABSTRACT

Formal specifications are important but often unavailable. Furthermore, writing these specifications is time-consuming and requires skills from developers. In this work, we present Deep Specification Miner (DSM), an automated tool that applies deep learning to mine finite-state automaton (FSA) based specifications. DSM accepts as input a set of execution traces to train a Recurrent Neural Network Language Model (RNNLM). From the input traces, DSM creates a Prefix Tree Acceptor (PTA) and leverages the inferred RNNLM to extract many features. These features are then forwarded to clustering algorithms for merging similar automata states in the PTA for assembling a number of FSAs. Next, our tool performs a model selection heuristic to approximate F-measure of FSAs, and outputs the one with the highest estimated F-measure. Noticeably, our implementation of DSM provides several options that allows users to optimize quality of resultant FSAs.

Our video demonstration on the performance of DSM is publicly available at **https://goo.gl/Ju4yFS**.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**;

KEYWORDS

Specification Mining, Deep Learning

ACM Reference Format:
Tien-Duy B. Le, Lingfeng Bao, and David Lo. 2018. DSM: A Specification Mining Tool using Recurrent Neural Network Based Language Model. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264597

1 INTRODUCTION

Formal specifications play important roles to the reliability and maintainability of software systems. Manually writing formal specifications can be very expensive and difficult as developers must have requisite skills and experiences. Furthermore, specifications can be quickly outdated due to rapid evolution of software systems. Thus, there is a need of automated solutions to infer specifications.

To help developers reduce the cost of manually drafting formal specifications, many automated approaches have been prosed [1, 4, 7, 8, 10]. One of the popular specification mining algorithms is to infer finite-state automaton (FSA) based specifications from execution traces [7, 8]. Nevertheless, the quality of mined specifications is not perfect yet, and more works need to be done to make specification mining better. For example, FSAs inferred using the k-tail algorithm are usually inaccurate for execution traces containing methods that frequently co-occur in particular orders, but are not required to occur exactly in these orders.

In this work, we implement a prototype tool based on our new specification mining algorithm [9] that performs deep learning on execution traces. We name the tool **DSM**, which stands for Deep Specification Miner. This tool takes as input a set of execution traces and performs several processing steps that eventually result in one FSA. First, DSM performs deep learning on the input execution traces to train a Recurrent Neural Network Language Model (RNNLM) [13]. Then, DSM constructs a Prefix Tree Acceptor (PTA) from the execution traces and leverage the learned language model to extract a number of interesting features from PTA's nodes. These features are then input to clustering algorithms for merging similar states (i.e., PTA's nodes). The output of an application of a clustering algorithm is a simpler and more generalized FSA that reflects the training execution traces. Finally, based on the predicted values of F-measure of constructed FSAs, DSM selects the one with highest predicted value of F-measure as output. Our evaluation on 11 target library classes shows that DSM has a good performance, i.e, an average F-measure of 71.97%.

The remainder of this paper is structured as follows. Section 2 highlights the bird's-eye view design of DSM. Sections 3 illustrates how DSM works with specific examples as well as demonstrates command line options provided by DSM. We presents our evaluation of DSM and discuss related works in Section 4 and Section 5, respectively. Finally, we conclude and mention future work in Section 6.

2 OVERALL DESIGN

Figure 1 shows the design of DSM. Our tool accepts as input a set of execution traces, which are sequences of methods. The output of DSM is a finite-state automaton (FSA) that reflects interactions between program methods in the input sequences. In our design, there are five processes in DSM: Recurrent Neural Network Based Language Model (RNNLM) learning, trace sampling, feature engineering, clustering, and model selection.

*Lingfeng Bao is a join first author. He was affiliated with Singapore Management University, Singapore when this work was performed.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264597

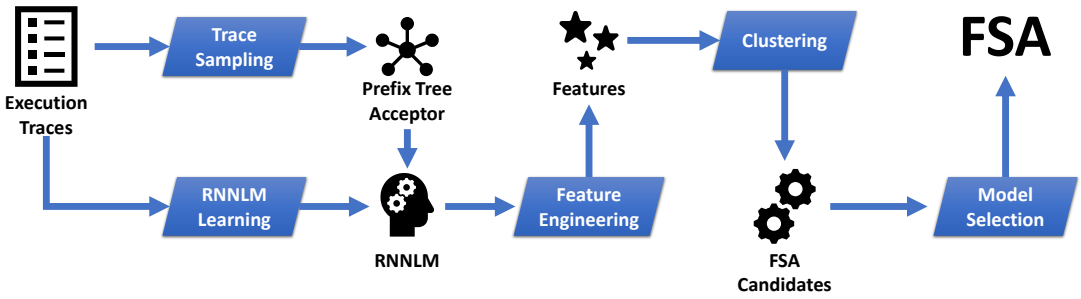

Figure 1: Overall Design of DSM

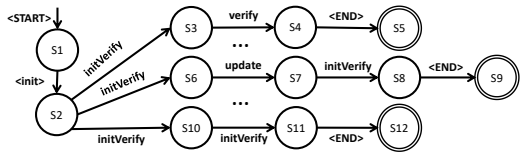

Figure 2: An Example Prefix Tree Acceptor (PTA)

RNNLM Learning: DSM accepts as input execution traces in the form of method sequences. Each of these sequences begins and terminates with two special symbols: <START> and <END>, respectively. These symbols are used for separating two distinct sequences. DSM provides utilities to select the underlying neural network architecture and configure learning parameters for training RNNLMs from execution traces. DSM supports the following three architectures: standard Recurrent Neural Network, Long Short-Term Memory (LSTM) [6, 16], and Gated Recurrent Units (GRU) [3].

Trace Sampling: Since it is expensive to use all sequences in the training data to construct FSAs, DSM leverages a heuristic to select a subset of method sequences. The goal here is to create a smaller subset that is likely to represent the whole set of all traces reasonably well. In particular, DSM's heuristic tries to find a subset of traces that covers all co-occurrence pairs[1] of methods in all training traces. More detail of DSM's trace sampling heuristic is available in our research paper [9].

Feature Extraction: DSM constructs a Prefix Tree Acceptor (PTA) from method sequences of the sampled execution traces. A PTA is a tree-like deterministic finite automaton (DFA) created by putting all the prefixes of sequences as states, and a PTA only accepts the sequences that it is built from. The final states of our constructed PTAs are the ones have incoming edges with <END> labels (see the step of RNNLM learning). Figure 2 shows an example of a Prefix Tree Acceptor (PTA).

Next, DSM extracts two different types of features based on PTA. The goal of this step is to provide sufficient information for clustering algorithms in the subsequent process to better merge PTA nodes. The following are the two feature types:

(1) **Type I**: This type of features captures information of previously invoked methods before the state S is reached. The values of type I features for state S is the occurrences of methods on the path between the starting state (i.e., the root of the PTA) and S. For example, according to Figure 2, the values of Type I features corresponding to node S_3 are: $F_{\text{<START>}} = F_{\text{<init>}} = F_{\text{initVerify}} = 1$ and $F_{\text{update}} = F_{\text{verify}} = F_{\text{<END>}} = 0$.

(2) **Type II**: This type of features captures the likely methods to be immediately called after a state is reached. Values of these features are computed by the inferred RNNLM in the deep learning step. For example, at node S_3 in Figure 2, `close` and <END> have higher probabilities than the other methods to be called afterward. Examples of type II features and their values for node S_3 output by a RNNLM are as follows: $P_{\text{initVerify}} = P_{\text{<END>}} = 0.4$ and $P_{\text{<START>}} = P_{\text{<init>}} = P_{\text{verify}} = P_{\text{update}} = 0.15$.

Clustering: DSM executes k-means [11] and hierarchical clustering [15] algorithms on the PTA's states with their extracted features. The goal of this step is to create a simpler and more generalized automaton that captures specifications of a target library class. Since both k-means and hierarchical clustering require the predefined input C for number of clusters, DSM tries with many values of C from 2 to `MAX_CLUSTER`[2] to search for the best FSA. Overall, the execution of clustering algorithms results in $2 \times$ (`MAX_CLUSTER` $- 1$) FSAs.

Model Selection: DSM employs a heuristic to select the best FSA among the ones output by the clustering algorithms using the input execution traces. It estimates Precision by first constructing a set P_I containing all pairs (m_1, m_2), where m_1 and m_2 appear consecutively (i.e., m_1 is called right before m_2) in the input execution traces. Then, DSM constructs another set P_M containing all pairs (m_1, m_2) that appear consecutively in a trace generated by an automaton M outputted by the clustering algorithms. The estimated Precision is: $\frac{P_I}{P_I \cup P_M}$. Next, DSM estimates the values of Recall by computing the percentage of all execution traces accepted by a given automaton M. Once all precision and recall of FSA models are estimated, DSM computes the expected value of F-measure (i.e., harmonic mean of precision and recall) for each of the automata. Finally, DSM returns the FSA with highest expected F-measure. More details of DSM's model selection heuristic are is available in our research paper [9].

3 USAGE SCENARIOS

In this section, we illustrate how DSM operates via several usage scenarios. Furthermore, we describe a number of DSM's key options to tune the accuracy of output FSA.

3.1 Scenario I – Simple Command

DSM is implemented in Python 3.5 with Tensorflow 0.12.0 and scikit-learn 0.19.1. We choose Python due to its simplicity and

[1] (m_1, m_2) is a co-occurrence pair if m_1 and m_2 appear together in at least one trace.

[2] By default, `MAX_CLUSTER=20`.

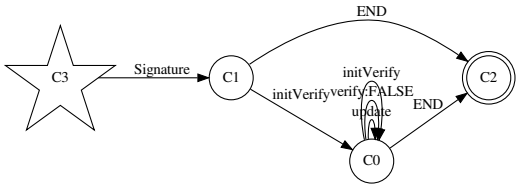

Figure 3: Resultant FSA for `java.security.Signature`.

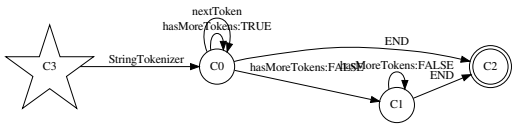

Figure 4: Resultant FSA for `java.util.StringTokenizer`.

convenience when deploying machine learning libraries. In a nutshell, DSM is a command-line application that allows users to input the path of data folder (i.e., "`--data_dir`") where execution traces are stored in "`input.txt`" file in the folder. DSM accepts traces that contains sequences of methods; for example, input traces of `java.security.Signature` is available at https://goo.gl/kaHQHd. These sequences can be collected from execution of software systems that are known to utilize the target API libraries or APIs. Additionally, test case generation tools (e.g., Randoop [14], etc.) can also be used to generate a richer set of test cases that capture many behaviors of the target libraries/APIs.

Users can run the following simple command to mine FSAs:

```
python3 DSM.py --data_dir [data folder]
```

After this command is executed, DSM performs several processing steps including learning a Recurrent Neural Network Language Model (RNNL), using its default parameters. Depending on learning configuration of RNNLM (e.g., number of hidden layers, size of each RNN layer, etc.), DSM might takes longer than 5 minutes to output a FSA. The resultant model mined by DSM is shown in two formats: text and image. Note that Graphviz[3] (i.e., Graph Visualization Software) must be installed to enable DSM to produce DOT files. Figure 3 shows one simple FSA mined by DSM.

In our implementation, "DSM.py" is the program main point that contains code portions calling five major modules of DSM (see Section 2). Advanced users who are interested in adapting our tool can investigate DSM's source code starting from "DSM.py". The latest version of DSM is developed and tested on Linux platform. We believe DSM can be employed on other platforms such as Windows and Mac OS as long as dependency libraries, especially Tensorflow, are properly installed. Details of DSM's installation instructions can be found at https://goo.gl/k3w8Hd.

3.2 Scenario II – Tuning DSM's Parameters

Assuming we are interested to infer a FSA for an input execution traces of `java.util.StringTokenizer`, which is available at https://goo.gl/myQLXo. Since it is possible that execution time and accuracy of resultant FSAs are not as expectation of users, DSM provides several parameters to tune its performance. Table 1 highlights a number of key parameters of DSM. In order to adjust the learning of RNNLM, users can change the configuration of the learned RNNLM such as number of hidden layers (i.e., "`--num_layers`"), number of states in each hidden layer (i.e., "`--rnn_size`") or type of RNN architecture (i.e., "`--model`").

In DSM's implementationm, we leverage the multi-cores of a CPU to speed up tasks that can execute in parallel such as feature engineering and clustering processes. To maximize the degree of parallelism in DSM, users can change the values of "`--max_cpu`" to indicate the number of processes that can be used by DSM. The more CPUs are assigned to DSM, the faster the specification mining process is. DSM also provides "`--min_cluster`" and "`--max_cluster`" options to optimize the clustering process (see Section 2). These options regulate the number of clusters considered by the clustering algorithms. By default the range is [2, 20]. The following is an example command that executes DSM with custom setting:

```
python3 DSM.py --data_dir [data folder] \\
 --num_layers 2 --rnn_size 32 --model lstm\\
 --min_cluster 3 --max_cluster 10\\
 --max_cpu 4
```

Figure 4 shows the resultant FSA inferred by DSM for `java.util.StringTokenizer`'s input traces (available at: https://goo.gl/myQLXo) using the above setting.

Table 1: DSM's key parameters

Parameter	Description
`--rnn_size`	Number of RNN states in each hidden layer
`--num_layers`	Number of hidden layers
`--model`	RNN type (i.e., RNN, LSTM, GRU)
`--batch_size`	Minibatch size
`--seq_length`	RNN sequence length
`--num_epochs`	Number of epochs
`--min_cluster`	Minimum number of clustering settings
`--max_cluster`	Maximum number of clustering settings
`--max_cpu`	Maximum number of CPUs can be used

Table 2: Target Library Classes. "#M" represents the number of class methods that are analyzed, "#Generated Test Cases" is the number of test cases generated by Randoop, "#Recorded Method Calls" is the number of recorded method calls in the execution traces, "NFST" stands for `NumberFormatStringTokenizer`.

Target Library Class	#M	#Generated Test Cases	#Recorded Method Calls
`ArrayList`	18	42,865	22,996
`HashMap`	11	53,396	67,942
`Hashtable`	8	79,403	89,811
`HashSet`	8	23,181	257,428
`LinkedList`	7	13,731	4,847
`NFST`	5	15,8998	95,149
`Signature`	5	79,096	205,386
`Socket`	21	80,035	130,876
`StringTokenizer`	5	148,649	336,924
`StackAr`	7	549,648	13,2826
`ZipOutputStream`	5	162,971	43,626

4 EVALUATION

In our experiments, we select 11 target library classes as the benchmark to evaluate the effectiveness of our proposed approach.

[3] https://www.graphviz.org/

Table 3: Effectiveness of DSM. "F" is F-measure.

Class	F (%)	Class	F (%)
ArrayList	22.21	Signature	100.00
HashMap	86.71	Socket	54.24
HashSet	76.84	StackAr	74.38
Hashtable	79.92	StringTokenizer	100.00
LinkedList	30.98	ZipOutputStream	88.82
NFST	77.52%	**Average**	71.97

These library classes were also investigated by previous research works [7, 8]. Table 2 shows further details of the selected library classes including information of collected execution traces. Among these library classes, 9 out of 11 are from Java Development Kit (JDK); the other two library classes are `DataStructure.StackAr` (from Daikon project) and `NumberFormatStringTokenizer` (from Apache Xalan). Table 3 shows the F-measure of DSM for the eleven target library classes. From the table, our approach achieves an average F-measure of 71.97%. Noticeably, for `StringTokenizer` and `Signature`, DSM infers models that exactly match ground truth models (i.e., F-measure of 100%). There are other 6 out of the 11 library classes where our approach achieves F-measure of 70% or higher.

5 RELATED WORK

k-tails is a classic algorithm proposed by Biermann and Feldman [2] to infer a FSA from execution traces. The algorithm takes as input a set of execution traces and a parameter k. *k-tails* first builds a prefix tree acceptor (PTA) , and then merges every two states of the PTA that have identical sequences of the next k method invocations (i.e., k-tails). The effectiveness of *k-tails* depends the choice of k and the quality of its input traces.

Krka et al. propose a number of specification miners that are capable of inferring a FSA from execution traces and likely invariants [7]. These miners are CONTRACTOR++, state-enhanced k-tails (SEKT), and trace-enhanced MTS inference (TEMI). Among the above three approaches, CONTRACTOR++ only utilizes value-based program invariants inferred by Daikon [5] to construct FSAs. On the other hand, state-enhanced k-tails (SEKT) and trace-enhanced MTS (Modal Transition System) inference (TEMI) analyze both execution traces and Daikon's likely invariants to infer FSA based specifications.

SpecForge is built on the top of Krka et al's proposed miners [7] and k-tails [2]. There are two important processes in SpecForge: model fission and model fusion. SpecForge employs model fission to extract many temporal rules from input FSAs, and select a number of interesting rules that frequently appear in input FSAs. Then, model fusion is utilized to construct a new automata that satisfies that selected temporal rules.

Additionally, Lo et al. propose SMArTIC that mines a FSA from a set of execution traces [4] using a variant of k-tails to construct a probabilistic FSA. Walkinshaw and Bogdanov propose an approach that allows users to input temporal properties to support a specification miner to construct a FSA from execution traces [17]. Mariani et al. propose k-behavior [12] that creates an automaton by inspecting one single trace at a time. Synoptic infers three kinds of temporal invariants from execution traces and uses them to generate a concise FSA [1].

Different from DSM, none of the above-mentioned studies leverage deep learning to mine specifications from execution traces. We have also shown in our research paper [9] that DSM substantially outperforms a number of baselines in terms of quality of mined specifications and ability to detect malicious behaviors of Android apps.

6 CONCLUSION

In this work, we present DSM which is an automated tool that applies deep learning to mine finite-state automaton (FSA) based specifications from execution traces. In particular, we describe the design, usage scenarios, and performance of DSM. DSM is open-source and can be run from command line with simple options. Our dataset, DSM's implementation, and a technical report that includes additional results are publicly available at: **https://github.com/lebuitienduy/DSM**.

ACKNOWLEDGMENT

This research was supported by the Singapore National Research Foundation's National Cybersecurity Research & Development Programme (award number: NRF2016NCR-NCR001-008).

REFERENCES

[1] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. of FSE*. ACM, 267–277.
[2] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.* 100, 6 (1972), 592–597.
[3] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* abs/1412.3555 (2014).
[4] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proc. of FSE/ESEC*. 265–275.
[5] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45.
[6] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[7] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proc. of FSE*. 178–189.
[8] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing Specification Miners through Model Fissions and Fusions (T). In *Proc. of ASE*. 115–125.
[9] Tien-Duy B Le and David Lo. 2018. Deep specification mining. In *Proc. of ISSTA*. ACM, 106–117.
[10] David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic steering of behavioral model inference. In *Proc. of FSE*. 345–354.
[11] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA., 281–297.
[12] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Trans. Software Eng.* 37, 4 (2011), 486–508.
[13] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*.
[14] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proc. of ASE*. Lawrence, KS, USA, 23–32.
[15] Lior Rokach and Oded Maimon. 2005. Clustering Methods. In *The Data Mining and Knowledge Discovery Handbook*. 321–352.
[16] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proc. of NIPS*. 3104–3112.
[17] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring Finite-State Models with Temporal Constraints. In *Proc. of ASE*. 248–257.