Textile & Leather Review

ISSN 2623-6281| www.tlr-journal.com | 10.31881/TLR

Deep Specification Mining Based on Transformer and Model Checking of Invariants

Jian Qi, Yiming Fan, Lili Qi

How to cite: Qi J, Fan Y, Qi L. Deep Specification Mining Based on Transformer and Model Checking of Invariants. Textile & Leather Review. 2026; 9:2022-2046. https://doi.org/10.31881/TLR.2026.2022

How to link: https://doi.org/10.31881/TLR.2026.2022

Published: 25 April 2026

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

Deep Specification Mining Based on Transformer and Model Checking of Invariants

Jian Qi[1]*, Yiming Fan[1], Lili Qi[2]

[1]School of Cyber Security and Computer, Hebei University, Baoding 071002, Hebei, China

[2]College of General Education, Yanshan College Shandong University of Finance and Economics, Jinan 271100, Shandong, China

*qijian1570321@163.com

Article

https://doi.org/10.31881/TLR.2026.2022

Published 25 April 2026

ABSTRACT

Software specifications play an important role in the software design process and are of great significance for improving software quality. Especially in the textile industry, with the rapid development of intelligent manufacturing, the complexity of control software for automated weaving machinery and textile production lines is increasing, making the formal verification of these systems essential. The finite state automata (FSA) generated by existing automatic specification mining methods still have insufficient accuracy in handling large-scale systems. In order to improve the accuracy of FSA, this paper proposes a deep specification mining method based on invariants and model checking. In this method, firstly, we use the Transformer model to generate feature values. Then, based on the different data types of the target library, different clustering algorithms were selected to cluster and generate the initial FSA. Afterwards, the optimal FSA of the F-measure was chosen, and the invariant formula was constructed by mining the information of potential invariants. Finally, refine the FSA through model checking. The refinement process is achieved by adding or removing transformation rules and states on the overall FSA based on the invariant formula. The proposed method provides a reliable solution for mining the behavioral specifications of complex logic systems in textile machinery, ensuring the stability of industrial control software. The effectiveness of our method was evaluated through 11 target library classes, and experimental results showed that the average F-measure of our proposed method can reach 97.06%, which is 25.09% higher than the average F-measure of the DSM.

KEYWORDS

software specification, finite state automata, transformer, model checking, textile industrial software

INTRODUCTION

With the continuous update of software, non-standard software project management has caused inconsistencies in many software codes and documents [1,2]. Faced with this challenge, software professionals seek sophisticated tools forassistance. Such tools typically feature functions like automatically generating code comments and extracting specifications from source code. Their core purpose is to improve program comprehension and enhance readability for developers. This need is particularly evident in industrial automation sectors, such as the textile industry, where the integration of cyber-physical systems (CPS) requires higher standards for software reliability and behavioral consistency in spinning and weaving automation. The software specification should be a document that can completely and accurately state the behavior of the software. The interfaces between the various components of a software product are very complex. Moreover, the internal structure of the software is constantly changing as it goes through various stages of software development. Therefore, it is difficult to describe the interface relationships of different components at different stages of development. Besides, there may be some problems such as contradiction, ambiguity, incompleteness, and confusion of abstract level when writing system specifications in natural language [3].

To address the limitations of such informal methods, researchers have incorporated mathematics into the software development lifecycle and developed rigorous mathematics-based formal methods. Formal methods enable seamless transitions across different software engineering activities and provide a rigorous mechanism for high-level validation, making them an ideal modeling tool for software development.

For example, finite-state automaton (FSA) describes the sequence of method calls through different states and transitions between states. Automata models offer a higher degree of precision compared to natural language descriptions [4,5].

As software specifications attract growing attention from software designers, techniques for mining software specifications are also emerging incessantly. The software specification mined by formal methods can be expressed in the form of FSA [6-11]. And also can be expressed in various formats of formulas [12-18]. Such as regular expressions, Linear Temporal Logic (LTL), Propositional Projection Temporal Logic (PPTL) and Computing Tree Logic (TCL) [19].

Software specifications modeled by finite state automata (FSA) provide a holistic overview of software systems, visually verify the responses of such systems to various external events, and identify latent software errors that are difficult to detect during actual operation. The K-tail specification mining technology only involves execution traces and does not involve invariants. This method considers any two methods in the model. If the K methods behind these two methods are the same, then the current method and the subsequent K methods were merged in turn.

There are many improved techniques based on K-tail [20-22]. One of these techniques is the GK-tail [23].This method can automatically generate Extended Finite State Machines (EFSMs) from interaction traces. The EFSMs model combines the classical algorithm for FSA generation with Daikon, which can capture the constraints of data values and the interaction between software com ponents. Beschastnikh I [8] proposed Synoptic, which can construct a trace model by specifying a set of regular expressions to extract multiple traces from the logs. And extracting three kinds of invariants from the traces. Next, using the invariants to refine and coarsen the trace model. During the process, always keep the model satisfying all invariants. The tool provides a clear view of hard-to-find errors by directly examining the logs.

Texada is a technique for mining invariant formula, which takes as input a user-defined property template and execution traces, and outputs a set of property instances that satisfy all traces [12].For example, an invariant specification $G(x \rightarrow XFy)$ means that “x always followed by y”. In addition, Texada also provides two control components, supporting threshold and confidence threshold. This tool combines these two indicators to evaluate the credibility of attributes.

Among the many FSA based methods for mining software specifications, the one that attracts our attention most is the Deep Specification Miner (DSM) [6].DSM use Recurrent Neural Network (RNN) language model to mine FSAs of software specifications and has significantly improved the quality of the software specifications it mines compared to previous methods. However, for textile-related software with multi-threaded tasks and complex state transitions, such as real-time monitoring systems for fabric defects or automated loom control, the extraction of precise specifications remains a bottleneck.

Specifically, DSM use a rich set of traces to train RNN, which effectively extracts feature values, and then use clustering algorithms to generate FSA, thereby capturing the system's specifications more concisely. DSM greatly improve the quality of mining software specifications.

To enhance the quality of FSAs, MCLSM takes the improvement of clustering algorithm as the starting point [24]. Fan, Y.; Wang, M. et al. [24] proposed a new method MCLSM, which use the ordering points to identify the clustering structure (OPTICS) clustering algorithm to cluster the feature values to improve the clustering effect. And the average F-measure of FSA generated by MCLSM reaches 92.19%, which is higher than DSM and most related tools.

However, two key issues remain unresolved for the DSM. First, RNN is incapable of parallel computation, and its long-term dependency issue results in unstable quality of the FSAs it expresses. Second, using FSAs to represent software specifications inevitably gives rise to state merging errors. To address the aforementioned issues with the DSM, we adopted the Transformer model in place of the RNN. We leveraged the self-attention mechanism of the Transformer model to mitigate the limitations of the RNN, which lacks parallel computation capability. This mechanism effectively resolves the problems of gradient vanishing and exploding to which RNNs are susceptible when processing long sequences. It also yields a notable improvement in the model's training speed. After that, we further refine FSA by extracting invariant information and constructing invariant formulas to check the model and solve the problem of erroneous merging between states in FSA. The experimental results show that the average $\mathrm{F-measure}$ of FSA mining by our proposed method can reach 97.06%.

Our article's contribution mainly lies in the following three aspects.

1. We propose a specification mining method based on Transformer and model checking of invariants. We use the Transformer model to overcome the shortcoming of RNN and generate invariant formulas based on invariants to refine FSA and reduce erroneous merging between states in FSA.

2. Implemented a method for mining software specifications using deep learning techniques and invariant formulas for model checking, and the tool is named: DSM-IM (Deep specification miner based on invariants and model checking).

3. The tool is evaluated on 11 target library classes. The experimental results showed that the average $\mathrm{F-measure}$ reaches 97.06%, which is higher than other related tools.

METHOD

Transformer Model

The Transformer model is a deep learning model framework based on attention mechanism, which was proposed by Google [25]. Transformer has a wide range of applications in natural language processing and other fields, and almost all of them have outstanding performance. The Transformer model structure is shown in Figure 1. It consists of an encoder and a decoder, and introduces an attention mechanism [26]. The encoder encodes the input data sequence into a continuous vector representation, and the decoder generates the next output based on the encoder's output and the previously generated output sequence. The most core component of the Transformer model is its self-attention mechanism, which allows the model to focus on other positions in the input sequence while processing information from each position, enabling it to capture dependencies in long sequences. On this basis, the multi-head self-attention mechanism captures richer semantic information through parallel computing.

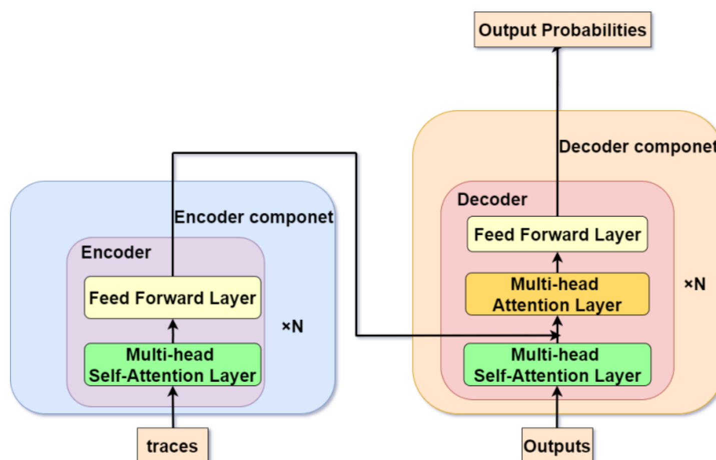


Figure 1. The Transformer model

Invariants and Model Checking

The invariant is a mathematical term published in 1993. This word is used in many contexts, and it means a constant. In the program, the invariant represents the property that the program state is always true at a certain point or multiple points. It can describe the relationship of all variables in the form of formulas at a certain position of the program (such as the exit point or entry point of a method in a class in Java).For a

simple example, the analysis of the entry point of the method m of a certain class draws a conclusion: a = 3 ∗ b + 2 ∗ c, which means that no matter what kind of test program is used to instantiate this class, the variables a, b, c always satisfies a = 3 ∗ b + 2 ∗ c at the entry point of method m.

Model checking is an automatic validation technique primarily used to verify whether a finite state system satisfies predefined properties. The basic idea is to explore all possible states of the system through automatic traversal, checking whether each state satisfies specific properties. In this article, we refine FSA based on invariants and model checking methods.

DSM

DSM is a technique for mining software system FSA specification through deep learning methods. DSM is a representative method in the field of automatic mining of software specifications, providing a more efficient and accurate solution to the problem of automatic mining of software specifications. Its main working principle is:

(1) Input the traces and train the model based on them: take the execution traces as input, input them into the neural network language model for training and learning, and learn the behavioral patterns and rules of the software system.

(2) Extract prefixes of the traces and calculate the feature values: Extract common traces prefixes based on the input execution traces, and use the trained neural network language model to extract feature values from the traces subset based on the traces prefixes.

(3) Constructing FSA with clustering algorithm: Cluster the obtained feature values using the K-means algorithm and merge similar states to construct an FSA.

(4) Selection and output of the model: Select the optimal FSA based on the evaluation criteria $\mathrm{F-measure}$.

SPECIFICATION MINING BASED ON TRANSFORMER AND MODEL CHECKING OF INVARIANTS

In this section, we will provide a detailed introduction to our proposed method. and the overall process is shown in Figure 2. The general process is divided into the following steps:

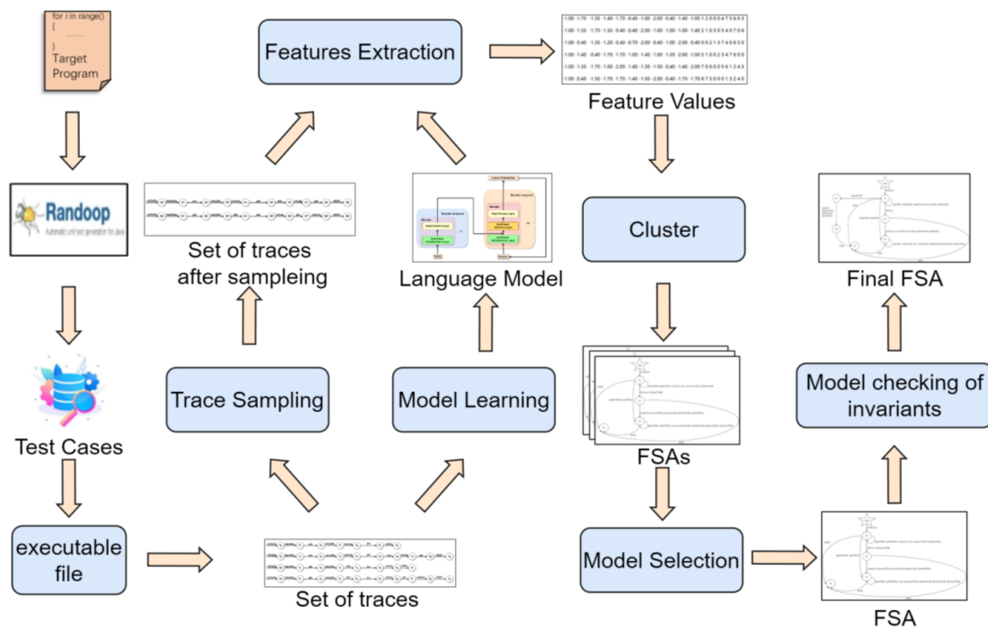


Figure 2. Overall flow chart

This paper use Randoop to generate a large number of test cases and execute these test cases to generate a rich set of execution traces that are used for extracting trace subsets, traces prefixes, and constructing Transformer models. Then, train the Transformer model to calculate and extract feature values, input clustering algorithm to obtain FSA as the initial model. Repeat the above process to generate multiple FSA as the initial model.

According to the model selection algorithm, screen the initial model and select the optimal FSA from it.

Then, use Texada to obtain invariant rules for execution traces. Using invariant rules to establish invariant formulas and algorithms. We use invariant formulas and algorithms to check the model, refine the initial model by filtering and removing illegal state transitions. During the checking process, FSA is not split, but rather filtered and removed from illegal state transitions directly on the model, and new states are added in a timely manner. This processing method ensures that FSA is always aggregated during the processing and avoids accidental deletion operations, making the model's expression more accurate.

The FSA Was Constructed

We take HashMap as our example program, and it has seven public methods:

HashMap(),put(),entrySet(),remove(),get(),size(),values().

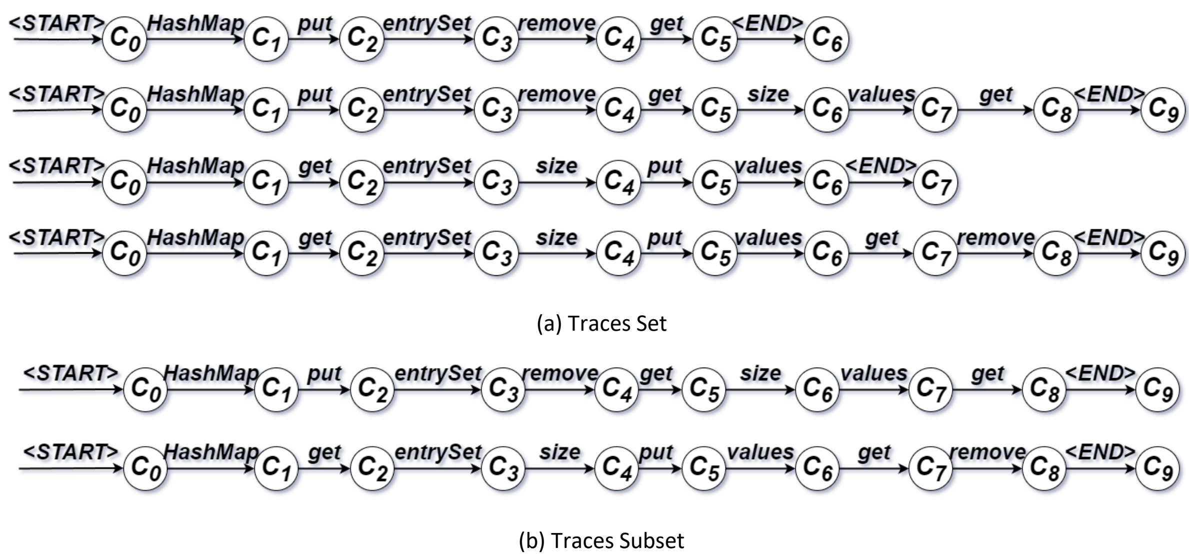
Trace generating This paper use Randoop, a framework for generating test cases for JAVA unit tests, which automatically generates test cases for compiled JAVA bytecode based on the Junit format [9,27]. By executing test cases, we can obtain a large number of traces to form a trace set. A trace can be represented as $\mathrm{tr} = < \mathrm{f}_1, \mathrm{f}_2, \ldots, \mathrm{f}_\mathrm{n} >$,where $\mathrm{f}_\mathrm{i} (1 \leq \mathrm{i} \leq \mathrm{n})$ is a method call.

Transformer model learning For the training of the Transformer model, we need to construct a training set of traces that includes multiple independent traces. Each trace starts with<START>< END>serves as the endpoint, representing the beginning and end of the trace respectively. These traces need to cover various usage scenarios of the software, from short sequences to long sequences, from simple operation sequences to complex behavioral patterns. After ensuring that the training set meets the above conditions, the training set is input into the Transformer Model shown in Figure 1 for model training to learn the rules and features behind these traces, thereby obtaining a Transformer model based on execution traces. The main function of this model is to predict the next method based on the method prefix of the current trace. For example, let's assume that a method trace is tr=<<START>, HashMap, entrySet, remove, <END>>.If the method prefix of the trace given in our training set is<START>, HashMap, the next method for prediction is entrySet. If the method prefix of its trace is<START>, HashMap, entrySet, the next predicted method is removed. Keep predicting until the appearance of<END>.

Trace Sampling We use traces subset to improve the efficiency of mining specifications. The traces subset should be able to coverall adjacent method pairs that have appeared in the execution traces. As shown in Figure 3, it is assumed that the execution traces are composed of four traces in Figure 3(a), and all adjacent method pairs in these four traces are:

{(<START>,HashMap),(HashMap,put),(put,entrySet),(entrySet,remove),(remove,get),(get,<END>),(get,size),(size,values),(values,get),(HashMap,get),(get,entrySet),(entrySet,size),(size,put),(put,values),(values,<END>),(values,get),(get,remove),(remove,<END>)}.

However, we can use the two traces in Figure 3(b)to cover all the method pairs that have appeared in Figure 3(a). Therefore, we can use a heuristic algorithm to achieve this operation.



(a) Traces Set

(b) Traces Subset

Figure 3. Traces Set and Traces SubSet

Feature extraction In order to provide more basis for clustering algorithms to cluster feature values, we use a combination of two types of feature values.

First, using the Transformer model, according to the predicted trace prefix $tr_o^i = < f_1', f_2', \ldots, f_i' >$,we can get a vector $P_{tr_o^i} = < P_{m_1}, P_{m_2}, \ldots, P_{m_k} >$,where $m_1, m_2, \ldots, m_k$ is all the methods appearing in the traces $P_{m_{1=i}} (1 \le r \le k)$ is the possibility of method m_r appearing after tr_o^i.After logarithmic processing of the probability vector,it is taken as the first type of feature value.

The second type of feature value depends on whether the method appears in the prefix tr_o^i.Using a value p to distinguish whether it has occurred or not. If the method m_r does not appear in the prefix tr_o^i,the value is p; If it does, the value is 1 − p.

For example, assuming that the predicted trace prefix is: tr_o^6 =<<START>, HashMap,put,entrySet, remove,get>,the output of the round obtained through the Transformer language model is a probability vector: $P_{tr_o^6}$ =<0.01, 0.02, 0.02, 0.03, 0.10, 0.02, 0.03, 0.07, 0.20, 0.40, 0.10>.Each of these components corresponds to a method and its probability, that is: $P_{<START>}$ =0.01, $P_{HashMap}$ =0.02, $P_{containsKey:TRUE}$=0.02,$P_{containsKey:FALSE}$=0.03,P_{values}=0.10,$P_{entrySet}$=0.02,P_{remove}= 0.03,P_{put}= 0.07,P_{get}= 0.20,P_{size}= 0.40,$P_{<END>}$= 0.10.

After performing logarithmic operation on the vector, we obtain the first type of feature value: $\log(P_{tr_o^6})$=<−2,−1.70,−1.70,−1.52,−1,−1.70,−1.52,−1.15,−0.70,−0.40, −1>.According to the position of

each method in the trace prefix, we can get: $F_{<START>} = 1$, $F_{HashMap} = 2$, $F_{containsKey:TRUE} = 0$,$F_{containsKey:FALSE} = 0$,$F_{values} = 0$,$F_{entrySet} = 4$,$F_{remove} = 5$,$F_{put} = 3$, $F_{get} = 6$, $F_{size} = 0$,$F_{<END>} = 0$. So, the feature value of the second type is <1,2,0,0,0,4,5,3,6,0,0>.Combine these two types of feature values into a eigenvalue vector:<−2,−1.70,−1.70,−1.52,−1,−1.70,−1.52,−1.15,−0.70, −0.40,−1,1,2,0,0,0,4,5,3,6,0,0>.

Clustering processing In this process, we adopt three clustering algorithms,namely K-means, hierarchical, and DBSCAN for clustering feature values.

From the preceding section, we extract the feature values of all method prefixes, and cluster them using the K-means, hierarchical, and DBSCAN algorithms. Prior to clustering, we first define the maximum number of clusters, which is denoted as max_cluster. For K-means and hierarchical clustering algorithms, these two algorithms both produce max_cluster − 1 FSAs respectively [28,29]. No matter how much max_cluster is set to, DBSCAN will get one FSA [29].So we obtained many FSA models, and Figure 4 shows an example of FSA that we presented.

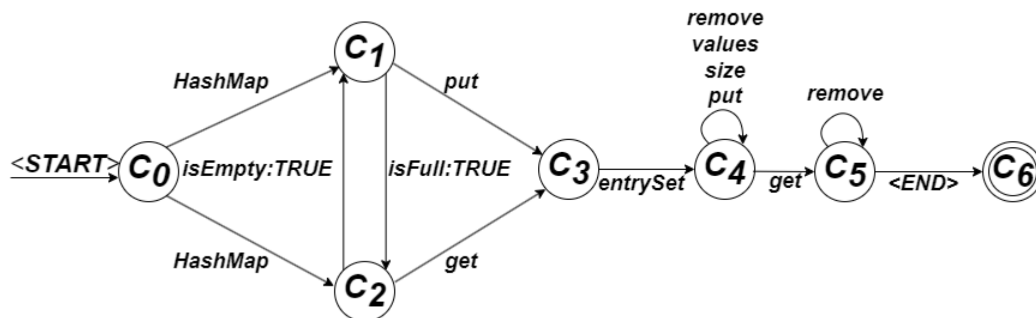


Figure 4. The FSA model generated by clustering

Model Selection

The FSAs generated by clustering algorithms need to screen out the optimal result based on its F1-measure value.

Regarding the measure for model selection, DSM and many other tools often choose a final FSA by evaluating Precision and Recall of all generated FSAs.

We use the ratio of |MPTR ∩fsa_pairs(F)| to |fsa_pairs(F)| as Precision.

F is the FSA to be evaluated, MPTR represents all method pairs that appear in the trace set TR, and fsa_pairs(F) represents method pairs appearing in F.

Note that we call a method pair (x, y) appears in F if x and y are labeled at two adjacent edges in F, respectively. In fact, we should consider the traces in TR as positive samples and that the traces in F are classified into positive samples. However, the number of the traces in F usually cannot be calculated. Thus, we consider the method pairs in TR as positive samples and that the method pairs in F are classified into positive samples. |fsa_pairs(F)| are the data that are classified as positive, and |MPTR ∩fsa_pairs(F)| are the data that are actually positive in the classification.

We use the ratio of |accepted_traces| to |TR| as Recall.

|accepted_traces| represents the number of traces in the trace set that the generated FSA can accept, and |TR| represents the number of all execution traces.

Using just one of Precision or Recall to evaluate an FSA cannot comprehensively evaluate the advantages and disadvantages of the FSA. Higher Precision means an FSA accepts fewer traces that should not be accepted, while higher Recall means an FSA can accept more traces that should be accepted. Thus, we combine Precision and Recall to obtain F1−measure (Equation 1) as the actual scoring criteria. It is worth noting that F1 − measure here is used to evaluate the optimal FSA during the experimental process.

$$F_1 - \text{measure} = 2 \times \frac{\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}} \tag{1}$$

The FSA generated by clustering algorithms may have the problem of state merging errors. So the initial model needs to be refined after model selection.

Model Checking of Invariants

Extract Invariant Formula

In order to refine the overall specification by utilizing the partial specifications (invariants) of the software, the finite state automaton generated in Section “The FSA was constructed” and filtered in Section “Model

selection" is used. This section will input the trace set, attribute template, and some initial values generated in the previous section into Texada to generate invariant formulas. The specific process is shown in Figure 5.

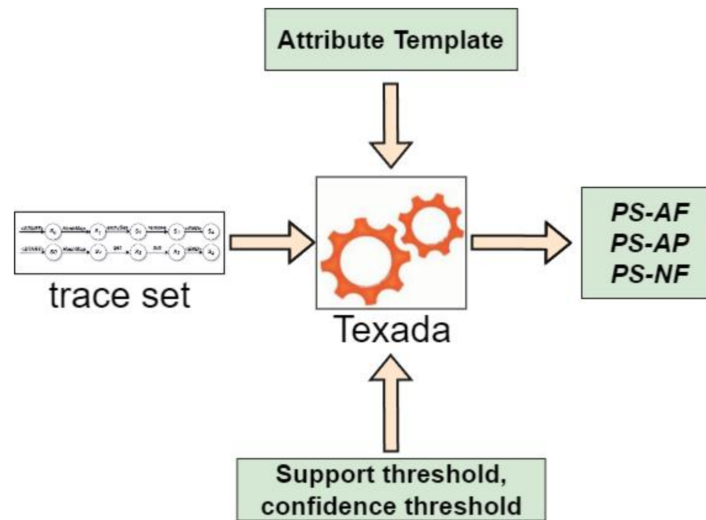


Figure 5. Generating invariant formulas

The generated invariant formulas are as follows:

(1) PS-NF(a,b): Event b will never occur immediately after event a occurs, expressed in the invariant formula as $G(a \rightarrow X(\neg b))$.

(2) PS-AP(a,b): Event a must immediately follow Event b. expressed in an invariant formula as $F(a) \rightarrow (\neg a \cup (b \wedge Xa))$.

(3) PS-AF(a,b): Event a must occur immediately after event b, which is expressed in the invariant formula as $G(a \rightarrow Xb)$.

Refine FSA

We use model checking to refine FSA. During the process of model checking, we use invariant formulas as restrictive rules to detect and correct the model.

As we shown in Table.1, prior to the refinement, we first define the variables to be employed in this process. The variables used in the refinement process are defined as follows:

(1) S and S' Representing FSA before and after refinement, respectively.

(2) K is a set of invariant formulas.

(3) Position is a variable used to mark the position of an element.

(4) Method denotes a method set, node denotes a node set. num is node index.

(5) Num is a parameter used for counting.

(6) Processed is used as an intermediate variable to store data information about FSA during the execution process.

Assuming set {method} = {method1,method2,method3,method4,method5,method6}.

Assuming set {node} = {node0,node1,node2,node3,node4}.

The corresponding specific algorithms and detailed explanations of the three invariant formulas are shown in Table 1. Taking PS − NF (a,b) as an example, illustrate the workflow of Table .1.

In set {method},

method1 = [<START>,ArrayList,toArray,remove:TRUE,<END>],

method2 = [<START>,ArrayList,isEmpty:TRUE,remove:TRUE,<END>],

method3 = [<START>,ArrayList,isEmpty:TRUE,addAll:TRUE,<END>],

method4 = [<START>,ArrayList,isEmpty:TRUE,indexof,<END>],

method5 = [<START>,ArrayList,toArray,addAll:TRUE,<END>],

method6 = [<START>,ArrayList,toArray,indexof,<END>].

Set {node}={C0,C1,C2,C3,C4}.

Table 1. Refinement Algorithm

Algorithm 1: Refinement	
	Input: S,K,Position,num,Processed,Method,node
	Output: *S'*
1	Initializing *S'* = *S*,*node*,*method*,*num* = 0;
2	While method & node ≠ Ø do
3	If $method_{num}$ violate *PS-NF(a,b)* then
4	*Position* ←Find the location of *a* in $method_{num}$;
5	$Processed_1$ ← [$node_{num}$[Position],C_{num},a];
6	*num* = *num* +1;
7	$Processed_1$ ← Nodes of all *method* in *S* after *a* are changed;
8	$Processed_1$ ← [new start node,new end node,b];
9	S ← [Original start node,original end node,a];
10	S'← $Processed_1$;
11	If $method_{num}$ violate PS-AP(a,b) then

12 *Position* ← Find the location of *b* in $method_{num}$;
13 $Processed_2$ ← [$node_{num}$[Position],C_{num},a];
14 $Processed_2$ ← [C_{num},C_{num+1},b];
15 *num* = *num* +1;
16 $Processed_2$ ← Nodes of all *method* in *S* after *b* are changed;
17 S ← [Original start node,original end node,b];
18 S'← $Processed_2$;
19 If $method_{num}$ violate PS-AF(a,b) then
20 *Position* ← Find the location of *a* in $method_{num}$;
21 $Processed_3$ ← [$node_{num}$[Position],C_{num},a];
22 *num* = *num* +1;
23 $Processed_3$ ← Nodes of all *method* in *S* after *a* are changed;
24 $Processed_3$ ← [start node,C_{num},b];
25 *num* = *num* +1;
26 S ← [Original start node,original end node,a];
27 S'← $Processed_3$;
28 Break;
29 Output *S'*;
30 end

The initial FSA in the above example is shown in Figure 6(a). The textual description is S = {[C0,C1,ArrayList],[C1,C2,b],[C1,C2,a],[C2,C3,d],[C2,C3,c] ,[C2,C3,e],[C3,C4,<END>]}.

Invariant formula:
PS-NF(*b*,*d*)

a:*toArray*
b:*isEmpty:TRUE*
c:*addAll*:*TRUE*
d:*remove:TRUE*
e:*indexof*

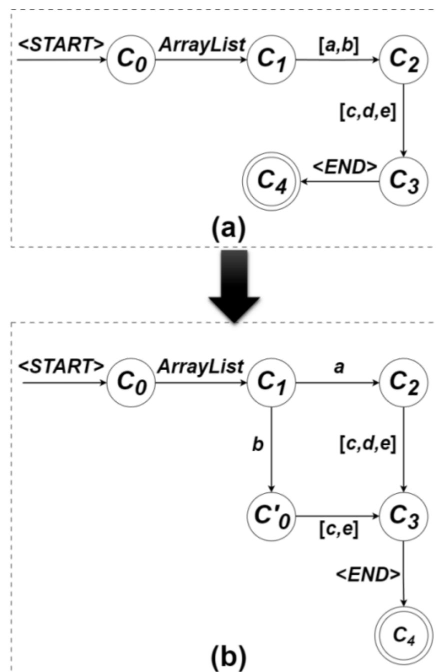


Figure 6. The process of model refinement is realized by PS-NF(b,d). (a): The initial FSA; (b): The refined FSA

The illegal state transition in this example is (b,d) in method2. Performing a deletion operation on an object determined to be empty is invalid and non-executable.Therefore the state transition violates PS−NF(a,b).Record this as: PS−NF(b,d).Therefore,execute PS−NF(a,b) in the algorithm 1.

First, we find the location of b in {method}(line 4). Then, find all{method}and{node}containing b, where{method}={method2,method3,method4}, {node}= {node2,node3,node4}.

After that, we construct a node C_0' for b. And change all nodes containing b (lines 5-7). We record the new nodes of{method}and the original nodes of{method}in the following form: [start node, end node, method]. Besides, we also recorded new data as:

Processed1={[C0,C1,ArrayList],[C1,C_0',b],[C_0',C3,d],[C_0',C3,c],[C_0',C3,e],[C3,C4,< END >]}(line 8).Every time a new node is added,it queries whether the original node is in Processed1 to ensure that the automaton is always aggregated(line 9).Finally,delete PS−NF(b,d).The refined FSA after the above operation is shown in Figure 6 (b).

Our processing method is not directly applied to the original automaton. This method is designed to avoid losing valid paths caused by the removal of erroneous state transitions. As shown in the example, if [C2,C3,d] was directly removed from the original automaton description, the correct path method1 will be lost. This will lead to a decrease in the accuracy of FSA.

PS −AP(a,b): First, we find the location of b and find all{method}and{node}containing b (line 12). Next, we construct a node for b, at the sametime, create a new node for a before b and change the node pointers of all method containing b (line 13).In this way, all paths containing b must first pass through a(lines 14-16).We record the new nodes of{method}and the original node of{method}in the following form:[start node,end node,method]. The newly added node is recorded in Processed2(line 17). Whenever a new node is added, the algorithm checks whether the original node exists in Processed2, ensuring the automaton remains consistently aggregated. Finally, delete the description of b in S.

PS −AF(a,b): First, we find the location of a and find all{method}and{node}containing a (line 20). Next, we create a new node for a and change the node pointers of all method that pass through a(lines 21-23).We record the new node of{method} and the original node of {method} in the following form:[start node,end node,method].The newly added node is recorded in Processed3(line 24).Every time a new node is added, it queries whether the original node is in Processed3 to ensure that the automaton is always aggregated. Then

we add b and the corresponding node to our newly segmented path, and add it before the <END> at the end (lines 25-26). Finally, we removed the description of a in S.

After processing each time invariant, we update the automaton file once to avoid missing path violations. Output refined FSA (lines 18,29). After the above process, we obtained a refined FSA and then used the most standard evaluation method F − measure to assess the accuracy of FSA.

EXPERIMENTS

In this section, we show the data and various settings used in our experiments, as well as the comparative analysis of this work and related work.

Data Set

The dataset used in our artical consists of 11 JAVA target library classes as shown in Table 2.To evaluate the effectiveness of DSM-IM, the 11 target libraries shown in Table 2 were used to validate DSM-IM. Table 2 shows the information of these classes, where NFST stands for NumberFormatStringTokenizer. The following is a brief introduction to 11 target libraries: ArrayList is a linear table data structure that supports random access to dynamic arrays. Due to inevitable state merging and illegal state transitions in FSA, it may encounter issues such as array out of bounds and concurrent operations. LinkedList is a linear table of a doubly linked list, and a possible issue in FSA is frequent calls to the get function. HashSet is a collection implementation based on HashMap, which ensures the uniqueness of stored elements. The possible issue in FSA is the inconsistency between HashCode () and equals (). HashMap, the hash table, which allows null values, may have issues with concurrent modifications leading to a dead loop. Hashtable is a hash table implementation that sacrifices performance to ensure thread safety, and this target library does not allow the occurrence of null values.

Table 2. Target Library Classes

Target Library Class	M	Recorded Method Calls
ArrayList	16	22996
LinkedList	8	4847
HashSet	9	257428
HashMap	10	67942
Hashtable	9	89811
Signature	6	205386
Socket	20	130876
ZipOutputStream	4	43626
StringTokenizer	6	336924
StackAr	8	132826
NFST	4	95149

Signature is a library in JAVA that provides digital signature algorithms for applications. Its function calls need to be made in the order of initSign (), updata () and sign (). Socket is a network communication endpoint used to encapsulate TCP/IP information. Possible issues may include not displaying closed ports and textual descriptors, or not handling timeouts and interrupts.

ZipOutputStream is a compressed output stream used to maintain a compression buffer, and a possible issue is that file entries are not written in order. StringTokenizer is a memory-efficient string tokenizer designed for string segmentation, yet it may suffer from issues such as redundant instantiation and incorrect resetting of tokenization positions. StackAr is a stack structure, and there may be issues where the push function is called when the stack is empty or the pop function is called when the stack is full.

NFST(NumberFormatStringTokenizer) is an abstract class for formatting numbers, which may have issues with multi-threaded shared instances and formatting string errors.

According to the analysis of the data in Table 2, these 11 target libraries include three types of data scales: small, medium, and large, and their data structures are also different. Therefore, using these target libraries

as datasets can be used to evaluate the effectiveness of DSM-IM on data of different sizes and complexities. And these 11 target library classes also used in previous work [6,7,24,30].

Experiment Setting

1. Transformer model construction: We build the Transformer model in Tensorflow 1.15.
2. Clustering: Setting max_cluster to 15, Both the K-means and hierarchical clustering algorithms require the elimination of one meaningless clustering result, and thus generate 14 FSAs respectively. DBSCAN clustering algorithms will get one FSA. So, in total, we can get 2 ∗ (max_clusters − 1) +1 = 29 FSAs.

Experiment Results and Analyses

In order to evaluate the quality of our model, we use 1-tail, SEKT 1,CONTRACTOR++,TEMI, DSM and MCLSM, six dynamic FSA inference techniques, as the benchmark.

1-tail is to select k value as 1 in the K-tail method [4].SEKT 1 is to select k value as 1 in the SEKT method [7]. The reason of choose k value as 1 is that 1-tail and SEKT 1 are the best performing methods among their respective approaches. CONTRACTOR++ and TEMI are also classic methods [7]. DSM is the first method to use deep learning technology[6].MCLSM is an improved clustering algorithm based on DSM [24]. As mentioned earlier, our proposed method is DSM-IM.

According to the different strategy divisions, the above methods can be divided [7]. DSM and K-tail belong to the Traces only strategy, CONTRACTOR++ belongs to the Invariants-only strategy, SEKT ,MCLSM and DSM-IM belong to the Invariant enhanced-traces strategy, and TEMI belongs to the Trace-enhanced-invariants strategy.

Our experimental results are listed in Table 3. In Table 3, “CONTR++” is CONTRACTOR++,“SEKT 1” is SEKT 1-tail, “-” means that the result is not available.

Table 3. F-measure (%)

Tools / class	1-tail	SEKT 1	CONTR++	TEMI	DSM	MCLSM	DSM-IM
ArrayList	13.96	36.03	13.07	16.87	22.21	81.94	89.06
LinkedList	27.15	86.02	24.52	7.51	30.98	98.49	95.24
HashSet	20.88	52.22	21.27	23.34	76.84	94.67	97.76
HashMap	25.41	68.94	-	-	86.71	97.02	100.00
Hashtable	42.39	92.78	-	-	79.92	91.42	99.75
Signature	61.54	66.88	63.98	39.06	100.00	86.44	100.00
Socket	35.89	55.15	28.37	-	54.24	100.00	97.32
ZipOutputStream	46.36	62.80	-	-	88.82	91.80	100.00
StringTokenizer	52.88	21.30	-	-	100.00	94.19	96.55
StackAr	16.54	34.91	16.54	-	74.38	96.55	98.43
NFST	24.57	30.40	25.78	11.80	77.52	100.00	93.54
Average	33.42	55.22	27.65	19.72	71.97	92.19	97.06

The average F − measure value of DSM-IM is 63.64%, 41.84%, 69.41%, and 77.34% higher than 1-tail, SEKT1, CONT++, and TEMI, respectively.

Compared to DSM, it can be found that the average F − measure value of DSM-IM has increased from 71.97% to 97.06%. This indicates that our method DSM-IM has significantly improved performance compared to DSM. Moreover, the LinkedList class has the highest degree of improvement from 22.21% to 89.06%.

Compared to the latest method MCLSM, DSM-IM performs better on most target libraries, and the average F − measure value of DSM-IM 4.87% higher than MCLSM. The most significant one is the performance on the signature target library, where our method achieved 100%, which is 13.56% higher than MCLSM.

Among the 11 target library classes evaluated, the F-measure values of three of these classes have achieved 100.00%, which are HashMap, Signature, and ZipOutputStream.

In order to verify whether the performance improvement of the experimental results is significant, we evaluated the results using paired t-test method. The evaluation results are shown in Table 4. DSM-IM is

significant compared to MCLSM and very significant compared to other methods. Overall, DSM-IM has statistical significance at a 95% confidence level relative to the baseline.

Table 4. T-test

Method	1-tail	SEKT 1	CONTR++	TEMI	DSM	MCLSM
t	14.982	6.419	13.547	21.194	3.547	2.737
p	0.000**	0.000**	0.000**	0.000**	0.005**	0.021*
Cohen's d	4.517	1.935	4.085	6.39	1.069	0.825

* p<0.05 ** p<0.01

CONCLUSIONS

This article proposes a specification mining method based on Transformer invariants and model checking, which utilizes Transformer's self-attention mechanism to capture input/output global dependency features. By extracting program invariants, invariant formulas are established and used for model checking, minimizing the impact of illegal state transitions and achieving FSA refinement. Experiments have shown that this framework has achieved a break through improvement in the accuracy of specification expression compared to traditional DSM models, with an average F − measure improvement of 25.09% on 11 target library classes, enabling FSA to more accurately express specification relationships and providing strong support for the precise description of software specifications. Our contribution lies in: for the first time, combining the self-attention mechanism of Transformer with formal verification methods, overcoming the inherent shortcomings of RNN in handling long sequence dependencies, establishing a dynamic optimization mechanism for FSA based on invariant formula constraints provides a more interpretable FSA for automated software validation. Specifically, the precision improvement of 25.09% in F-measure demonstrates the potential of our method in modeling complex control logic for textile machinery, such as high-speed weaving and intelligent dyeing systems, where subtle state transition errors can lead to production defects.

However, the method of automatically mining software specifications based on deep learning still has certain limitations. This method only relies on execution traces, which means that a large set of traces that can cover

various situations are a necessary condition for ensuring FSA accuracy. If the trace coverage is insufficient, it may result in FSA not accurately reflecting the true specifications of the software, thereby affecting the accuracy and reliability of software verification. Subsequent research may consider introducing more diverse data sources or further optimizing the acquisition strategy of execution traces to further improve the stability and accuracy of FSA. Furthermore, we plan to evaluate the robustness of this framework in real-world textile manufacturing software environments to better support the digital transformation of the industry.

Author Contributions

Conceptualization, J.Q. and Y.F.; methodology, J.Q. and L.Q.; validation, Y.F.; investigation, J.Q. and Y.F.; writing—original draft preparation, J.Q. and Y.F.; writing—review and editing, J.Q. and L.Q.; visualization J.Q. and Y.F.; All authors have read and agreed to the published version of the manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

Funding

This research received no external funding.

Data Sharing Agreement

The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Acknowledgements

Not applicable.

REFERENCES

[1] Dagenais B, Robillard MP. Recovering Traceability Links Between an API and Its Learning Resources. Proceedings of the 34th International Conference on Software Engineering (ICSE); 2-8 Jun 2012; Zurich, Switzerland. Piscataway, NJ, USA: IEEE; 2012. p. 47-57. doi: 10.1109/ICSE.2012.6227207

[2] Medvidovic N, Taylor RN. Software Architecture: Foundations, Theory, and Practice. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2; 1-8 May 2010; Cape Town, South Africa. New York, NY, USA: ACM; 2010. p. 471–472. doi: 10.1145/1810295.1810435

[3] Dagenais B, Robillard MP. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering; 7-11 Nov 2010; Palo Alto, CA, USA. New York, NY, USA: ACM; 2010. p. 127-136. doi: 10.1145/1882291.1882312

[4] Biermann AW, Feldman JA. On the Synthesis of Finite-State Machines from Samples of Their Behavior. IEEE Transactions on Computers. 1972; 100(6):592-597. doi: 10.1109/TC.1972.5009015

[5] Cicchello O, Kremer SC. Inducing Grammars from Sparse Data Sets: A Survey of Algorithms and Results. Journal of Machine Learning Research. 2003; 4(Otc):603–632. Available from: https://jmlr.org/papers/v4/cicchello03a.html

[6] Le TDB, Bao L, Lo D. DSM: A Specification Mining Tool Using Recurrent Neural Network Based language Model. Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 27 Aug-1 Sep 2018; Stockholm, Sweden. New York, NY, USA: ACM; 2018. p. 896-899. doi: 10.1145/3213846.3213876

[7] Krka I, Brun Y, Medvidovic N. Automatically Mining Specifications From Invocation Traces and Method Invariants. Proceedings of the 36th International Conference on Software Engineering (ICSE); 31 May-7 Jun 2014; Hyderabad, India. New York, NY, USA: ACM; 2014. p. 178-189. doi: 10.1145/2635868.2635890

[8] Beschastnikh I, Brun Y, Schneider S, Sloan M, Ernst MD. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering; 9-16 Sep 2011; Szeged, Hungary. New York, NY, USA: ACM; 2011. p. 267-277. doi: 10.1145/2025113.2025151

[9] Li JG, Feng Q, Yang SQ, Li J, Wu B. Mining Program Workflow From Interleaved Traces. Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10); 25-28 Jul 2010; Seattle, WA, USA. New York, NY, USA: ACM; 2010. p. 613-622. doi: 10.1145/1835804.1835883

[10] Ohmann T, Herzberg M, Fiss S, Halbert A, Palyart M, Beschastnikh I, et al. Behavioral Resource-Aware Model Inference. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering; 15-19 Sep 2014; Västerås, Sweden. New York, NY, USA: ACM; 2014. p. 19-30. doi: 10.1145/2642937.2642988

[11] Stolz V, Bodden E. Temporal Assertions Using AspectJ. Electronic Notes in Theoretical Computer Science. 2006; 144(4):109–124. doi: 10.1016/j.entcs.2006.02.007

[12] Lemieux C, Park D, Beschastnikh I. General LTL Specification Mining. Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE); Lincoln, NE, USA. Piscataway, NJ, USA: IEEE; 2015. p. 81-92. doi: 10.1109/ASE.2015.71

[13] Zhang N, Yu B, Tian C, Duan Z, Yuan X. Temporal Logic Specification Mining of Programs. Theoretical Computer Science. 2021; 857(1):29-42. doi: 10.1016/j.tcs.2020.12.032

[14] Weimer W, Necula GC. Mining Temporal Specifications for Error Detection. Proceedings of the Tools & Algorithms for the Construction & Analysis of Systems, International Conference (TACAS), Held as Part of the Joint European Conferences on Theory & Practice of Software (ETAPS); 4-8 Apr 2005; Edinburgh, UK. Berlin, Germany: Springer; 2005. p. 461-476. doi: 10.1007/978-3-540-31980-1_30

[15] Yang J, Evans D, Bhardwaj D, Bhat T, Das M. Perracotta: Mining Temporal API Rules from Imperfect Traces. Proceedings of the 28th International Conference on Software Engineering (ICSE); 20-28 May 2006; Shanghai, China. New York, NY, USA: ACM; 2006. p. 282-291. doi: 10.1145/1134285.1134325

[16] Gabel M, Su Z. Javert: Fully Automatic Mining of General Temporal Properties From Dynamic Traces. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 9-14 Nov 2008; Atlanta, GA, USA. New York, NY, USA: ACM; 2008. p. 339-349. doi: 10.1145/1453101.1453150

[17] Gabel M, Su Z. Symbolic Mining of Temporal Specifications . Proceedings of the 30th International Conference on Software Engineering (ICSE); 10-18 May 2008; Leipzig, Germany. New York, NY, USA: ACM; 2008. p. 51-60. doi: 10.1145/1368088.1368096

[18] Gabel M, Su Z. Online Inference and Enforcement of Temporal Properties. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1; 1-8 May 2010; Cape Town, South Africa. New York, NY, USA: ACM; 2010. p. 15-24. doi: 10.1145/1806799.1806806

[19] Goranko V. Logic in Computer Science: Modelling and Reasoning About Systems. Journal of Logic, Language, and Information. 2007; 16(1):117-120. Available from: http://www.jstor.org/stable/40180443

[20] Lo D, Mariani L, Pezzè M. Automatic Steering of Behavioral Model Inference. Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software engineering; 7-11 Sep 2009; Florence, Italy. New York, NY, USA: ACM; 2009. p. 345-354. doi: 10.1145/1595696.1595761

[21] Mariani L, Pezzè M. Dynamic Detection of COTS Component Incompatibility. IEEE Software. 2007; 24(5):76-85. doi: 10.1109/MS.2007.138

[22] Walkinshaw N, Bogdanov K. Inferring Finite-State Models with Temporal Constraints. Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE); 15-19 Sep 2008; L'Aquila, Italy. Piscataway, NJ, USA: IEEE; 2008. p. 248-257. doi: 10.1109/ASE.2008.35

[23] Lorenzoli D, Mariani L, Pezzè M. Automatic Generation of Software Behavioral Models. Proceedings of the 30th International Conference on Software Engineering (ICSE); 10-18 May 2008; Leipzig, Germany. New York, NY, USA: ACM; 2008. p. 501-510. doi: 10.1145/1368088.1368157.

[24] Fan Y, Wang M. Specification Mining Based on the Ordering Points to Identify the Clustering Structure Clustering Algorithm and Model Checking. Algorithms. 2024; 17(1):28. doi: 10.3390/a17010028

[25] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017; 30:5998-6008. Available from: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[26] Bahdanau D, Cho K, Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate. International Conference on Learning Representations. 2015. doi: 10.48550/arXiv.1409.0473

[27] Robinson B, Ernst MD, Perkins JH, Augustine V, Li N. Scaling Up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs. Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE); 6-10 Nov 2011; Lawrence, KS, USA. Piscataway, NJ, USA: IEEE; 2011. p. 23-32. doi:10.1109/ASE.2011.6100059

[28] MacQueen J. Some Methods for Classification and Analysis of Multivariate Observations. Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability; 21 Jun-18 Jul 1965; Berkeley, CA, USA. Los Angeles, CA, USA: University of California Press; 1967. p. 281-297.

[29] Rokach L, Maimon O. Clustering Methods. The Data Mining and Knowledge Discovery Handbook. Boston, MA, USA: Springer; 2005. p. 321-352. doi: 10.1007/0-387-25465-x_15

[30] Le TDB, Le XBD, Lo D, Beschastnikh I. Synergizing Specification Miners Through Model Fissions and Fusions. Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE); 9-13 Nov 2015; Lincoln, NE, USA. Piscataway, NJ, USA: IEEE; 2015. p. 115-125. doi: 10.1109/ASE.2015.83