Towards Better Quality Specification Miners

David Lo and Siau-Cheng Khoo
Department of Computer Science, School of Computing
National University of Singapore
{dlo,khoosc}@comp.nus.edu.sg

Abstract

Softwares are often built without specification. Tools to automatically extract specification from software are needed and many techniques have been proposed. One type of these specifications – temporal API specification – is often specified in the form of automaton (i.e., FSA/PFSA). There have been many work on mining software temporal specification using dynamic analysis techniques; i.e., analysis of software program traces. Unfortunately, the issues of scalability, robustness and accuracy of these techniques have not been comprehensively addressed.

In this paper, we describe a framework that enables assessments of the performance of a specification miner in generating temporal specification of software through traces recorded from its API interaction. Our framework requires the temporal specification produced by the miner to be expressed as probabilistic finite state automaton (PFSA). The framework accepts a user-defined simulator PFSA and a specification miner. It produces quality assurance measures on the specification generated by the miner. We investigate metrics used in these measures by adapting techniques found in artificial intelligence, program analysis, bioinformatics and data mining to the software specification domain. Extensive experiments on two specification miners have been performed to evaluate the effectiveness of the proposed quality assurance measures.

1 Introduction

Presence of bugs and non-existence of specifications are common problems faced by software engineers. It is desirable if every program is specified formally. Unfortunately, difficulty in writing formal specification has proven to be a barrier in adoption of formal specification [1]. Worst yet, imprecise, changing requirements and short time to market [9] contribute to construction of programs with poor or no specification. The situation is further aggravated by the lack of specification or irrelevancy of specification during program evolution (cf. [13]).

These problems have been partially addressed by techniques which infer specifications from program code or protocols from call sequences, both statically and dynamically. Some of the dynamic inference techniques also enable testing and detection of semantic anomalies [40, 33].

Recently, there has been a surge in software engineering research to adopt machine learning and statistical approaches to address these problems, especially in the area of specification discovery [12, 15, 34, 1, 4, 14, 33, 39, 22]. In [16], Fox illuminates the use of machine learning to bridge the gap between high level abstraction expressing software engineering problems and low level program behaviors. He points out that some baseline models can be learned automatically to aid characterization and monitoring of system.

Along similar line of research, Ammons et al. coin the term specification mining as a machine learning approach to discover program specification by analyzing program execution traces [1]. Under the assumption that the program being mined must “reveal strong hints of correct protocols” during its execution, Ammons et al. demonstrate that correct specification can be obtained through their technique. Specifically, their technique focuses on mining of specification which reflects temporal and data dependency relations of a program through traces of its API-client interaction. The specification discovered models API-client interaction protocol, which is expressed initially as a probabilistic finite state automaton (PFSA). To reduce the effect of errors in training traces, transitions with low likelihood of being traversed can later be pruned. After pruning, the probabilities are dropped and an FSA is obtained.

Despite the proliferation of specification-mining research, there is not much report on issues pertaining to the quality of specification miners. Specifically, we note that issues such as scalability and robustness of miners, level of user intervention required during mining have not been comprehensively addressed. As an illustration, in [1], it was reported that “in order to learn the rule [i.e., automaton], we need to remove the buggy traces from the training set.” This indicates the problem with the limitation of choosing good
training sets. In a later work [2], it was noted that in order to debug specification generated by specification miner, it might be necessary to exhaustively inspect each of the traces, which can be hundreds or thousands in number.

Hence, there is a demand for a generic framework that can assess the quality of specification miners. Such a framework must address the issue of limited training sets as well as provide objective measures to the performance of specification miners. Performance should be measured in multiple dimensions: miners’ scalability, robustness and accuracy.

**Scalability** determines a specification miner’s ability to infer large specification. **Robustness** refers to its sensitivity to error present in the input data. **Accuracy** refers to the extent of an inferred specification being representative of the actual specification.

These measurements extend from the existing set of measurements found in the literature; specifically, the measurement of **accuracy** is supported by the measurement of recall and precision as provided by Nimmer et al. [29].

Together, these measurements do not only define the quality of specification miners in different dimensions, they also aid the design and development of new specification miners.

In this paper, we propose a generic framework for assessing the quality of automaton-based specification miners. For uniformity sake, our framework requires any specification miner under assessment to have the following input-output behavior: Let a program execution trace be a sequence of method calls to an API interface. Given a (multi-)set of program execution traces $T$, a minority of which might be erroneous, the specification miner infers sequencing/temporal constraints among the method calls in the form of a probabilistic finite-state automaton.

We do not constrain the automaton-based specifications to be deterministic; in fact, a miner is expected to perform its task in the presence of non-deterministic specification.

We choose to represent specification as a **Probabilistic FSA** (PFSA) instead of FSA, for the following reason: **Probabilities attached to a protocol specification enables the control of trace-generation process so that the collection of traces generated mimics some characteristics of the traces that can be collected from actual API interactions.** For example, sub-protocols within a protocol specification may be present more frequently than others in the actual interaction with API interface – analogous to the idea of hotspot found in program execution [21]. Such behavior can be made to exhibit in a set of generated traces through supply of appropriate probabilities at various transitions of a specification automaton.

It has been proven that learning an FSA from examples of sentences accepted by a language is not decidable [1, 17]. On the other hand, learning a PFSA from examples is decidable (cf. [10, 3]) though inefficient (cf. [24]). This theoretical finding has prompted Ammons et al. to use PFSA as an intermediate step to the learning of a FSA. In our framework, we believe that PFSA is essential to the capturing of sub-protocol behavior, and a good specification miner should uncover such sub-protocol behavior by producing a PFSA containing appropriate probabilistic transitions.

Our framework enables any specification miner with the required input-output behavior to be assessed under a simulated environment. The framework operates as follows: Given a specification miner, a PFSA and a percentage of expected error, the framework generates a multiset of traces from the PFSA with the specified percentage of erroneous traces. Running the specification miner against these traces will result in a mined PFSA. By comparing the behavior of the mined PFSA with that of the original PFSA, the framework can assess the accuracy of the mining as performed by the given miner.

Furthermore, by varying the percentage degree of expected error and the size of the original PFSA, the framework enables respective assessment of robustness and scalability of the miners.

We also propose several metrics for objective assessment of automaton-based specification miners, and devise corresponding techniques to support these metrics. Some of the concepts and techniques, such as automation alignment, are novel, while others are adapted from a wide spectrum of research, including simulation, software testing, bioinformatics, artificial intelligence, program analysis and data mining. Such a multi-disciplinary approach enables the best tool to be used in appropriate quality measurement.

We have built a prototype of our framework, and begun using it to assess some of the existing specification miners. In this paper, we describe our comprehensive experiments on two specification miners.

The outline of the paper is as follows: In section 2, a typical specification mining process is presented and our framework architecture is outlined. Sections 3 and 4 describe our solutions to two major issues related to quality assurance measurement; they are model-and-trace generation, and metrics and techniques for quality assurance. Section 5 describes specification miners used including our own in brief. Section 6 describes our experiments and results. We discuss related work and conclude in Section 8.

## 2 Framework Structure

In this section we’ll first discuss typical specification mining process with its limitation. Next, we’ll describe an overview of our framework which addresses the limitations of typical specification mining process.
Figure 1. Typical Specification Miner Process

Before presenting the framework structure, we describe a typical specification miner process, which is depicted in Figure 1. A miner’s input is typically in the form of API interaction traces, where each trace represents a sequence of method calls (with or without argument). Preprocessing is then performed on these traces to turn them into abstract representation of traces. Specification miner then learns from these traces to produce a specification. The specification can be expressed in various forms: automaton [1, 12, 39, 34, 4], algebraic equations [22], Hoare-style equation of pre and post-condition [15], etc. Human judgment is often employed at this stage to assess the accuracy of the mined specification and to evaluate the performance of the miner. Some systems, such as [1], in addition permits mined specification to be modified manually before feeding back to the specification miner again.

Other systems, such as Daikon [15, 29], assess the miner through its accuracy in recalling correct information (invariants) and in reducing generation of incorrect information. However, they fall short of providing systematic support for assessment of scalability and robustness of miners. It is clear that scalability and robustness are important determinants for the usability of miners; the former determines the limit of a miner in handling complex systems, and the latter determines the usefulness of a miner in handling mildly corrupted input.

2.1 Typical Specification Miner Process

2.2 Proposed Framework

Our proposed framework aims to address all the above quality assurance concerns. It accepts specification models of varying complexity, and generates sets of simulated traces that reflect the characteristics of those protocol specifications, including the presence of error. It then evaluates miner’s performance in recovering the original model from three dimensions: its accuracy, robustness and scalability.

The framework is shown in Figure 2. Its trace generator component generates traces based on a specification model in PFSA format. These simulated traces are then used to train the specification miner, culminating with a mined PFSA model. The original model and the mined model are then used by the specification miner quality assurance subsystem to generate various quality assurance metrics.

There are two major issues our framework need to address: (1) model and trace generation, and (2) quality assurance metrics and their techniques. These major issues will be discussed in sections 3 and 4 respectively.

3 Simulator Model & Trace Generation

In these section, our simulator model and trace generation methodology will be discussed. Simulator model govern the probability distribution of traces. Models can be specifically generated to test QA properties of interest. Trace generation methodology ensures generated traces are representative of the model under some coverage criteria.

3.1 Simulator Model

Our model is in the form of probabilistic finite state automaton (PFSA), an example of which is shown in Fig-
Figure 3. Sample Simulator Model

Figure 4. Model Generation Algorithm

3.2 Trace Generation

Actual program trace can be mapped to string of alphabets as shown by Ammons et al. through ‘standardization’ process [1]. Strings of alphabets generated by simulator can...
be considered as abstraction of actual program traces i.e. an alphabet representing a particular method call. Based on this abstraction, we generate traces as strings of alphabets.

Trace generation will generate two types of output - error and normal traces. A trace is defined as a sequence of transition names that forms a path leading from the start node to the end node of a PFSA. We define an error trace as one that includes a transition sinking at an error node.

Since normal traces are generated from a PFSA, we can determine the probability of a trace by multiplying together the probability of its constituents. We write \( p(t) \) to denote the probability of a trace \( t \).

We propose two algorithms for trace generation. They perform stratified random walk guided by the probability of PFSA’s transitions. Both algorithms ensure that highly probable traces (sentences) accepted by the PFSA model will statistically be more likely to appear in the multiset of generated traces. (We use the term “sentence” and “trace” interchangeably.) On the other hand, the two algorithms differ in the complexity in which they attempt to include all probable traces in the generation process.

The first trace-generation algorithm, TraceGen1, is akin to the “code and branch coverage” criterion used in generating program test cases [19, 7, 30]. Given a PFSA \( M \) and a global percentage of error, the algorithm generates a multiset of traces \( T \) possessing the following property:

**Property 1** For a sufficiently large \( T \), there is \( n > 0 \) such that all transitions in the PFSA \( M \) occurs at least \( n \) times in the traces of \( T \).

This property ensures that all transitions in \( M \) have the opportunity to be used for trace generation. The algorithm detail is depicted in Figure 5.

\( M_{EI} \) is the PFSA \( M \) with error \( EI \) injected. At program point (*), a trace is generated by starting from start node of the model and independently ”throwing a dice” at each node for decision on which transition to take according to the probability of the transitions until end node is reached. Traces generated will then reflect the probabilities of the transitions in the simulator model (i.e. distribution of generated traces is governed by the model).

Traces will continue to be generated until all transitions have been covered at least \( N \) times or MAX number of traces have been generated. We use \( N \) here rather than 1 to accommodate slower learner that requires more than 1 sentence in the language to infer the automaton model.

In [19], Gupta highlights the limitation of the standard “code and branch coverage” criterion in generating test cases: some combinations of branch decisions will not be executed. Correspondingly, some combinations of sequences of transitions may not have the chance to be used for mining.

---

**Procedure TraceGen1**

**Inputs:**
- \( M \) : Automaton model
- \( EI \) : Error injection
- \( N \) : Cover
- \( I \) : Maximum number of loop
- \( \text{MaxPopE} \) : Maximum possible error trace population
- \( \text{Max} \) : Maximum trace number
- \( GE \) : Global error injection probability

**Outputs:**
- A multiset of traces

**Method:**

```plaintext
let \( M_{EI} = M \cup EI \)
let \( E = \{ \text{list of all transitions in model } M \text{ identified by the transition name, its source and sink nodes} \}
let \( Errlist = \{ \text{list of all possible error traces from } M_{EI} \text{ bounded by } \text{MaxPopE} \text{ where for each, transitions in } M_{EI} \text{ are traversed at most } I \text{ times} \}
let \( Ctr = \{ \text{a map from } e \text{ in } E \text{ to a number} \}
\text{- initialized to 0} \)
do { \text{let } \text{rand} = \text{random number between 0 to 1} \}
if (\text{rand} < GE) { \text{let } TE = \{ \text{a trace generated randomly from } Errlist \}
Output TE
}
else { \text{let } T = \{ \text{a trace generated from } M \text{ (see text)} \}
Output T
let \( E' = \{ \text{all transitions traversed by } T \}
For each \( e' \in E' \text{ increase } Ctr[e'][t] \text{ by 1} \}
} \text{while (}\exists e \in E, Ctr[e] < N \text{ & number of traces } \leq \text{Max})
```

**Figure 5. Trace Generation Approach 1**

Our second algorithm, TraceGen2, aims to rectify this limitation. It generates a multiset of traces \( T \) which possesses the following property:

**Property 2** Given \( i > 0 \text{ and } n > 0 \), let \( K_i \) be the set of all paths in \( M \) linking start node to end node such that any transition of \( M \) cannot occur more than \( i \) times in any path. Then, for a sufficiently large \( T \) generated by TraceGen2, all paths in \( K_i \) occurs at least \( n \) times in the traces of \( T \).

Note that Property 2 is a stronger property than Property 1. The algorithm can be found in Figure 6.

In trace generation approach 2, a list of all possible paths – the number of which is bounded by \( \text{MaxPopN} \) – is generated from \( M \) such that each transition in \( M \) can occur at most \( I \) times in a trace. The probability of each such path can be calculated by multiplying the probabilities of the constituent transitions. Traces will then be generated.
The quality of a specification miner is measured along three dimensions: accuracy, robustness and scalability.

The accuracy of a specification miner is determined by its ability in recovering simulator models by learning the simulated traces, in the absence of error. This is usually measured via the notion of “recall” and “precision” [29]. However, it is not sufficient when the simulator model is in PFSA format, as we will elaborate shortly.

We define robustness of a specification miner as its ability in remaining accurate in recovering simulator models from simulated traces, in the presence of error. Thus, a robust miner should be able to filter erroneous traces in building mined models. As commonly observed, erroneous traces usually constitute a small proportion of the entire collection of traces.

Lastly, we define scalability of a specification miner as its ability in remaining accurate in recovering simulator models of varying sizes.

As these measurements are orthogonal, we can conveniently compose them, and objectively discuss about the robustness of a scalable miner, or the scalability of a robust miner.

4.1 Quality Assurance Metrics

For the sake of presentation, we denote a simulator model by X and a mined model by Y. We use the term “sentence” and “trace” interchangeably.

Central to our measurements is a thorough treatment of accuracy. In determining the accuracy of mined models in PFSA format, we propose four metrics of measurement.

Trace Similarity. The first two metrics measure similarity in terms of the number of sentences accepted by both X and Y. That is, the percentage of sentences generated by X that are accepted by Y, and vice versa. We denote them as TS.XY and TS.YX respectively. They are analogous to the notion of “recall” and “precision” – a standard measure from information retrieval (cf. [18, 37]) – respectively. Nimmer et al. further relate these two metrics to measures of completeness and soundness respectively [29].

Structural Similarity. Our third metric aims to measure structural similarity between X and Y. Structural similarity is important since needlessly complex automaton will inhibit users’ ability to understand the mined specification.

We measure structural similarity, denoted by SS, of X and Y by computing three sets of sub-automata: Sim captures the similarities between X and Y, and DifX (DifY) captures the differences between X (Y) and Sim. SS is then a normalized ratio of transitions in similarity and difference automaton:

\[ SS = \frac{\#Sim.Edges}{\#Sim.Edges + \#DifX.Edges + \#DifY.Edges} \]

where \( \#X.Edges \) denotes the number of transitions in X.

Probability Similarity. For models that are represented by PFSA, it is not sufficient to measure their similarity by simply examining their recall and precision. It is equally important to determine if both the simulator and the mined
models generate the same traces at similar frequencies, and thus place emphasis on similar sub-protocols. Thus, our final metric measures the similarity in terms of probabilities assigned to common traces generated by both $X$ and $Y$: A trace might possibly be generated by both $X$ and $Y$; however, its probability might differ greatly. This measurement is called co-emission.

Co-emission has been used in measuring similarity between two Hidden Markov Models. Lyngsø et al. propose several versions of similarity measurement [28, 27]. One such metric which is adopted here, denoted by $PS$, provides an unbiased and normalized similarity measurement of two models:

$$PS(X, Y) = \frac{2 \cdot P_{CE}(X, Y)}{P_{CE}(X, X) + P_{CE}(Y, Y)}$$

$$P_{CE}(X, Y) = \sum_{s \in L(X \cup Y)} (P_X(s)P_Y(s)).$$

Here, $P_{CE}(X, Y)$ denotes a co-emission probability, determining the probability that a sentence $s$ is generated by both $X$ and $Y$ independently. $P_X(s)$ and $P_Y(s)$ denote the probability of generating sentence $s$ by $X$ and by $Y$ respectively.

### 4.2 Quality Assurance Techniques

For each category of metrics we defined in subsection 4.1, we support them with a measurement technique. These are: Automaton Language Search, Automaton Alignment and HMM-HMM Comparison Based Technique.

**Automaton Language Search.** Here, we calculate the percentage of sentences generated by $X$ that are accepted by $Y$ and vice versa. The generated sentences will be a different set of samples drawn from $X$ than the one used for training $Y$. Splitting training and test sets ensures the learner under test avoids "overfitting" i.e. learns the training set so closely that it does not generalize well to original model [20]. Counter-examples – sentences generated by $X$ (or $Y$) that are not accepted by $Y$ (or $X$) respectively – will also be reported. The metrics that are supported by this method correspond to the TS.XY and TS.YX.

This technique is effective in measuring the quality of $Y(X)$ provided the set of traces generated are representative of $X(Y)$. To this end, we use the TraceGen1 procedure in Figure 5 to help in trace generation. This ensures polynomial time complexity with coverage assurance similar to "code and branch coverage" criterion. The effectiveness of the algorithm, in terms of complexity, is important in the situation where we wish to evaluate the scalability of a specification miner: It is important that the computation of quality assurance measurement be itself scalable as well.

**Automaton Alignment.** This method aims to generate automata that capture structural similarity and difference of $X$ and $Y$. Based on similarity and difference automata, we can then calculate the normalized ratio of transitions in similarity and differences automata. This number corresponds to SS.

The algorithm, as outlined in Figure 7, comprises five steps: (1) Provision of a standard representation of $X$ and $Y$ in string format, called $s_1$ and $s_2$ respectively. (2) Alignment of $s_1$ and $s_2$ using a variant of global alignment technique (cf. [36]). (3) Unification of two aligned strings to extract a similarity automaton. (4) Extraction of difference automata. The best alignment results are kept in nodesAlign[], where each element in the array contains two consecutive subsequences from $s_1$ and $s_2$ – possibly with some pads inserted to indicate no possible match of nodes – that represents a best alignment. As an example, assuming that some of the nodes in $X$ are named as $x_0, x_1, \ldots, x_5, x_{end}$ and some of those in $Y$ are named $y_0, y_1, \ldots, y_8, y_{end}$. A possible pair of alignments can be as follows:

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_4$</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_{end}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_0$</td>
<td>$y_2$</td>
<td>$y_1$</td>
<td>$y_4$</td>
<td>$y_3$</td>
<td>$y_7$</td>
<td>$y_5$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$y_6$</td>
<td>$y_{end}$</td>
</tr>
</tbody>
</table>

In this example, the “dash” (“-”) represents an added pad. Unification operation performed at Step 3 attempts to discover segments of similarity between automata $X$ and $Y$ by connecting pairs of aligned nodes to form similarity automata. From the pair of alignments kept in nodesAlign[], two pairs of aligned nodes, $(x, y)$ at some $j^{th}$ position and $(x', y')$ at some $k^{th}$ position, are unifiable if the following conditions hold: (1) $x$ is adjacent to $x'$ in $X$ and $y$ is adjacent to $y'$ in $Y$, and (2) at least one transition joining $x$ and $x'$ has the same label (i.e., method call) as the one that joins $y$ and $y'$. Successful unification results in a small automaton joining $x$ and $x'$. Repetitive application of unification returns a (possibly disjoint) automaton representing similarity between $X$ and $Y$.

**HMM-HMM Comparison Based Technique.** This technique computes probability similarity (PS). This is based on the work of Lyngsø et al. on comparison between two Hidden Markov Models [28, 27]. Figure 8 outlines the algorithm for calculating $P_{CE}$ between two automata. Probability similarity (PS) can be calculated from $P_{CE}$ as defined
in subsection 4.1.

Note that at each iteration of probability table update, we extend (in the worst case) the co-emission probability computation to another pair of nodes (each from X and Y respectively) which is only one distance further away from the start nodes. We approximate co-emission by ending the probability computation only after each of the possible loops is executed at least twice. This is ensured by repeating the probability table update for \(2 \times \min(|X.Edges|, |Y.Edges|)\) times. We refer readers to [27] for further interesting discussion about the behavior of the algorithm that our HMM-HMM comparison based algorithm adapt.

---

**Procedure AutomatonAlignment**

**Inputs:** X: First automaton, Y: Second automaton

**Outputs:**

- nodesAlign[]: best alignments of nodes in X with nodes in Y
- sim[][]: list of similarity automatons for each best alignment
- diffx[][]: list of difference automatons between sim and X for each best alignment
- diffy[][]: list of difference automatons between sim and Y for each best alignment
- qaMet[]: SS values for each best alignment

**Method:**

1. Let s1 = Enumeration of all nodes in X using breadth-first search of automaton nodes in ascending (transition name) order
2. Let s2 = Enumeration of all nodes in Y using the same techniques as that for generating s1
3. For each i indexing nodesAlign[] do
   - Extract similarity automaton into sim[i] by a series of unification
4. Extract difference automaton into diffx[i] and diffy[i] by including transitions in X and Y that are not in sim[i]
5. Let w = \(\max(|\text{diffx}[i].\text{Edges}|, |\text{diffy}[i].\text{Edges}|)\)
6. Let qaMet[i] = \#sim[i].\text{Edges} + \#sim[i].\text{Edges}

**Output:**

- nodesAlign[], sim[], diffx[], diffy[], qaMet[]

---

**Procedure HMM-HMM**

**Inputs:** X: First automaton, Y: Second automaton

**Outputs:**

- \(P_{CE}(X,Y)\): sum of co-emission probabilities for all sentences s commonly generated by X and Y

**Method:**

1. Let uX = Enumeration of all nodes in X
2. Let uY = Enumeration of all nodes in Y
3. Let n = number of nodes in X
4. Let m = number of nodes in Y
5. Let dProb[][] = Create table of size \(n \times m\)
6. Let xStart[] = corresponding rows representing start nodes of X according to uX
7. Let yStart[] = corresponding cols representing start nodes of Y according to uY
8. Let xStop[] = corresponding rows representing end nodes of X according to uX
9. Let yStop[] = corresponding cols representing end nodes of Y according to uY
10. Let iterNo = 2*min(|X.Edges|, |Y.Edges|)
11. For each (0 \(< i < \#xStart\)) and (0 \(< j < \#yStart\))
   - Initialize dProb[xStart[i]][yStart[j]] to 1
   - Iteratively update dProb[][] for iterNo times by doing the following at each iteration:
     - For each (0 \(\leq i < n\)) and (0 \(\leq j < m\))
       - Let dSum = 0
       - For each k where uX[k] has a transition \(t_k\) to uX[i] and each h where uY[h] has a transition \(t_h\) to uY[j]
         - If \(t_k\) and \(t_h\) have the same label, then
           - Let \(p(t_k)\) = probability of transition \(t_k\)
           - Let \(p(t_h)\) = probability of transition \(t_h\)
           - \(dSum += dProb[k][h] \times p(t_k) \times p(t_h)\)
           - \(dProb[i][j] = dSum\)
     - \(P_{CE}(X,Y) = \text{Sum of} \ dProb[xStop[i]][yStop[j]]\) for all i and j

**Figure 8. HMM-HMM Based Algorithm**

---

5 Specification Miners Used

In this section, the two specification miners used in our experiments will be briefly described. The two specification miners are sk-strings learner [31] used by Ammons et al., and our own miner (Merged Multiple Filtered Specification Miner – MMFSM). Full description of the specification miners’ algorithms are outside the scope of this paper.

5.1 sk-strings

The sk-strings algorithm is an extension of k-tails heuristic of Biermann and Feldman [6] for stochastic automata. In k-tails, two nodes are considered equivalent by looking at up to subsequent k-length strings that can be generated from
them. Different from k-tails, in sk-strings, the subsequent strings not necessarily end in an end node except for strings of length less than k. Furthermore, only top 5% of the most probable strings that can be generated from both nodes are considered.

The sk-strings algorithm first builds a canonical machine similar to a prefix tree acceptor from the traces. The nodes of this canonical machine are later merged if they are indistinguishable with respect to top 5% most probable strings of length at most k that can be generated starting from them.

5.2 Merged Multiple Filtered Specification Miner

Merged Multiple Filtered Specification Miner (MMFSM) comprises four major blocks – clustering, filtering, learning and merging, as shown in Figure 9. Filtering and clustering is meant to address the issue of robustness and scalability respectively.

Traces deviating from common trace population rules will be removed. The resultant filtered traces are then separated into multiple clusters whose number is determined automatically by considering similarities within each cluster and differences among clusters. By clustering common traces together, it is expected that the learner is able to learn better and over-generalization of a subset of traces is not propagated to other clusters. These cluster of traces will then be fed into a specification miner. The sk-strings learner described in previous paragraph will be used. Each cluster can be considered as an independent sub-protocols. If a combined view is desired, a merger sub-system will produce a merged automaton while ensuring the probabilities assigned to traces remains correct and no further generalization is made during the merging process. By default, we merged the automatons together.

6 Experiments

Three sets of experiments were conducted to evaluate the performance of two specification miners in terms of accuracy, robustness and scalability respectively.

The first experiment acts to illustrate in detail an accuracy experiment using a simple model. The latter, more extensive experiments were performed to measure robustness (a total of 600 experiments) and scalability (a total of 160 experiments). Since robustness and scalability are measured in terms of accuracy in presence of error and increased model size respectively, we indirectly measure accuracy through these experiment.

Off-the-shelf sk-strings learner [31] used by Ammons et al. and MMFSM was used as the user-defined specification miners for evaluation. Default parameter settings is used for sk-strings both when used stand-alone and within MMFSM. Since MMFSM is meant to improve scalability and robustness, only results pertaining to scalability and robustness will be shown.

6.1 Material

In the first experiment, we simulated the automaton generated by Ammons et al. in their analysis of X11 windowing library (cf. [1]) with addition of probabilities. This experiment measures the accuracy performance of sk-strings learner. The simulator model used is shown in Figure 10. Probabilities are distributed equally to transitions from the same source node (not shown in the diagram). We generated traces using trace generation method 2 which enumerated all combinations of transitions with parameters N and I set to 10 and a cap of maximum 10000 traces.

In the second experiment, we evaluated two learners’ performance in terms of robustness. We used similar model and transition probabilities as that used in experiment 1, but the model was without any non-determinism and repeat use of alphabet assigned to transitions. This is meant to produce a base model that can be learned perfectly. Error node and transitions were then injected to the automaton to conduct robustness tests. The model used with injection of errors (nodes and transitions labelled as Z) is shown in Figure 11.

We expect specification miner to be able to filter error. We compared the inferred automaton with the simulator model shown in Figure 11 without error nodes and transitions and recorded the similarity and difference metrics. We generated traces using trace generation method 1 with parameters N and I set to 10 and a cap of maximum 10000 traces. Four, eight and ten percents of error were injected to the system (i.e., 4, 8 and 10 percent of generated traces respectively will be erroneous). In each case, we ran a hundred experiments and recorded the average performance.
In the third experiment, we evaluated the scalability of the learners by generating distinct models of various sizes. Two sets of experiments were conducted, each with a different independent variable. In the first set of experiments, we varied the number of nodes (by 10, 20, 30, and 40) in the model and maintained the number of outgoing transitions per node to at most four. In the second set of experiments, we varied the number of outgoing transitions per node (by 3, 5, 7, 9) and maintained the number of nodes at 10. For each case, we performed 10 experiments and recorded their average performance. The models are generated based on properties and algorithms described in subsection 3.1.

We generated traces using trace generation method 1 with parameters $N$ and $I$ set to 10 and a cap at maximum 10000 traces. No error was injected to the system. Since we imposed a cap of 10000 traces, there might be a concern that training traces might not satisfy the coverage criterion for model of large size. This was not the case in our experiments, as only once did the cap was reached; for the other 159 experiments, coverage criterion was met first.

6.2 Experiment 1 Findings

Sk-strings inferred automaton – Figure 6.2 – is different from the simulator automaton (shown in Figure 10).

Language Search

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Transition</th>
<th>Probability</th>
<th>Probability†</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>G</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>C</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>B</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>E</td>
<td>0.5</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>E</td>
<td>0.5</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>D</td>
<td>0.5</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>D</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>H</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>END</td>
<td>F</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>END</td>
<td>E/I†</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† Used in Experiment 2.

Automaton Alignment

The best alignments of nodes is as follows:

<table>
<thead>
<tr>
<th>START</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>End0</td>
</tr>
</tbody>
</table>

Using language search method, 100% of the sentences generated by $X$ using trace generation method 2 were accepted by $Y$ (TS.XY = 1). Conversely, only 66.559% of the sentences generated by $Y$ were accepted by $X$ (TS.YX = 0.66559).

HMM-HMM Comparison Based Technique

The result using HMM-HMM comparison technique was 0.154 for $P_{CE}(X,Y)$ and 0.865 for PS. The metrics are defined in subsection 4.1.

Analysis

Trace similarity score indicated that the mined model $Y$ had almost 35% chance of generating traces not recognizable by the simulator model $X$. This points to a lack of precision in the mined automaton. Probability similarity score showed that the probability of generating common sentences is high (0.865 out of 1). Next, in comparing the two automaton structures (cf. Figure 13(b)), we noted particularly that the two transitions which led to nondeterminism were missing in the mined automaton. In place of these transitions, other new transitions were introduced. However, these new transitions are not equivalent with the original ones.

To investigate the cause of inaccuracy, we repeated the experiment by removing the two transitions from the simulator model that caused non-determinism. We found that
6.3 Experiment 2 Findings

Here, we evaluated the robustness of sk-strings learner and MMFSM. The experiment results are captured in Table 1 and Table 2 respectively. Column E% corresponds to the percentage of erroneous traces; other columns refer to QA metrics defined in subsection 4.1.

<table>
<thead>
<tr>
<th>E%</th>
<th>TS.XY</th>
<th>TS.YX</th>
<th>SS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.000</td>
<td>0.958</td>
<td>0.693</td>
<td>0.946</td>
</tr>
<tr>
<td>8</td>
<td>1.000</td>
<td>0.925</td>
<td>0.597</td>
<td>0.953</td>
</tr>
<tr>
<td>10</td>
<td>1.000</td>
<td>0.901</td>
<td>0.549</td>
<td>0.951</td>
</tr>
</tbody>
</table>

Table 1. sk-strings Robustness – Errors

<table>
<thead>
<tr>
<th>E%</th>
<th>TS.XY</th>
<th>TS.YX</th>
<th>SS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.000</td>
<td>0.996</td>
<td>0.962</td>
<td>0.947</td>
</tr>
<tr>
<td>8</td>
<td>1.000</td>
<td>0.990</td>
<td>0.925</td>
<td>0.947</td>
</tr>
<tr>
<td>10</td>
<td>1.000</td>
<td>0.982</td>
<td>0.888</td>
<td>0.945</td>
</tr>
</tbody>
</table>

Table 2. MMFSM Robustness – Errors

Analysis

For sk-strings, all traces generated by the simulator model X were accepted by the inferred model Y. On the other hand, we noted a drop in the acceptance of traces generated by Y. This drop is equivalent to the noise injected (4.2% vs. 4%, 7.5% vs. 8% and 9.9% vs. 10%); this indicates that the learner is unable to filter erroneous traces. Structural similarity is affected by injection of error as well. This worsens as we increase the level of error injection. We conclude that the sk-strings learner is not robust.

MMFSM is similar to sk-strings in that all traces generated by the simulator model X was accepted by the inferred model Y. Different from sk-strings, we noted only a slight drop in the acceptance of traces generated by Y. This drop is far less than the noise injected (0.4% vs. 4%, 1% vs. 8% and 1.8% vs. 10%); this indicates that the learner was able to filter erroneous traces. Structural similarity is also only slightly affected by the injection of error.

6.4 Experiment 3 Findings

We analyze the results from two sets of scalability experiments conducted. In the first set of experiments, we generated distinct models by varying no of nodes while keeping max transitions per node at 4. In the second set of experiments, we varied max no of transitions while keeping the total no of nodes constant constant at 10.
6.4.1 Number of Nodes

The result of varying number of nodes is shown in Table 3 and Table 4 for sk-strings and MMFSM respectively. Columns X.N and X.TN correspond to the number of nodes and maximum number of transitions per node in the simulator model.

<table>
<thead>
<tr>
<th>X.N/X.TN</th>
<th>TS.XY</th>
<th>TS.YX</th>
<th>SS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/4</td>
<td>0.998</td>
<td>0.292</td>
<td>0.192</td>
<td>0.460</td>
</tr>
<tr>
<td>20/4</td>
<td>1.000</td>
<td>0.006</td>
<td>0.064</td>
<td>0.022</td>
</tr>
<tr>
<td>30/4</td>
<td>1.000</td>
<td>0.003</td>
<td>0.045</td>
<td>0.019</td>
</tr>
<tr>
<td>40/4</td>
<td>1.000</td>
<td>0.006</td>
<td>0.022</td>
<td>0.026</td>
</tr>
</tbody>
</table>

Table 3. sk-strings Scalability - Nodes

<table>
<thead>
<tr>
<th>X.N/X.TN</th>
<th>TS.XY</th>
<th>TS.YX</th>
<th>SS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/3</td>
<td>0.998</td>
<td>0.514</td>
<td>0.207</td>
<td>0.574</td>
</tr>
<tr>
<td>10/5</td>
<td>0.980</td>
<td>0.489</td>
<td>0.055</td>
<td>0.632</td>
</tr>
<tr>
<td>10/7</td>
<td>0.976</td>
<td>0.507</td>
<td>0.049</td>
<td>0.634</td>
</tr>
<tr>
<td>10/9</td>
<td>0.976</td>
<td>0.515</td>
<td>0.056</td>
<td>0.625</td>
</tr>
</tbody>
</table>

Table 6. MMFSM Scalability - Max Transitions

6.4.2 Max Number of Transitions per Node

The result of varying maximum number of transitions per node is shown in Table 5 and Table 6 for sk-strings and MMFSM respectively. Columns X.N and X.TN correspond to the number of nodes and maximum number of transitions per node in the simulator model.

<table>
<thead>
<tr>
<th>X.N/X.TN</th>
<th>TS.XY</th>
<th>TS.YX</th>
<th>SS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/3</td>
<td>1.000</td>
<td>0.214</td>
<td>0.211</td>
<td>0.341</td>
</tr>
<tr>
<td>10/5</td>
<td>0.994</td>
<td>0.206</td>
<td>0.159</td>
<td>0.334</td>
</tr>
<tr>
<td>10/7</td>
<td>0.997</td>
<td>0.253</td>
<td>0.226</td>
<td>0.363</td>
</tr>
<tr>
<td>10/9</td>
<td>0.998</td>
<td>0.109</td>
<td>0.102</td>
<td>0.221</td>
</tr>
</tbody>
</table>

Table 5. sk-strings Scalability - Max Transitions

Analysis

The above results shows that both sk-strings and MMFSM were affected when we scaled-up the model size. Acceptance of traces generated by $Y$, structural similarity and probability similarity were adversely affected. Comparing the two set of experiments, the effect was less adverse when we increase the maximum number of transitions per node.

MMFSM was generally better in terms of acceptance of traces generated by $Y$ up to a factor of 27 (i.e. 20 nodes case). It was also better in terms of probability similarity up to factor of 12 (i.e. 20 nodes case). However, it was worse in terms of structural similarity up to a factor of 5 (i.e. 7 transitions case). The worsening performance in structural similarity was due to over-generalization of each sub-protocol at each cluster, causing difficulty in merging.

Comparing the results of the two set of experiments, the benefit of MMFSM was less in the second set of experiments (i.e. when we increase the maximum number of transitions). This was so since increasing the number of transitions while maintaining the number of nodes would result in a more ‘bushy’ automaton that is harder to separate during clustering.

7 Related Works

There have been numerous work in the research of specification mining. They can be classified into two groups, depending on how the mined specifications are represented: automaton-based [1, 12, 39, 34, 4] and non-automaton based [22, 14, 15, 33] specification mining.

The specification miner described in [1] has been extensively assessed in this paper. In other work, Whaley et al. extract object-oriented component interface sequencing constraints to form multiple finite state automatons [39]. Reiss et al. encode program execution traces as directed acyclic graph to aid visualization and understanding of program [34]. Arts et al. dynamically extract program models from Erlang program as state graph model for model checking and visualization [4]. We believe that these and other similar miners can equally be assessed under our framework with minimal changes.

In the field of data mining, Keogh et al. provide benchmark by providing diverse datasets for testing various time-series data-mining algorithms [25]. A significant difference between data mining and specification mining is the availability of data for learning – as the case in [1]. In the current work, we tacitly avoid this problem through simulation.

In the area of devising metrics for measurements, there have been similar work on comparison metrics in different domains, such as Bioinformatics and Artificial Intelligence domains. These include comparison of two gene sequence[36], comparison of two protein structures[5, 8], comparison of two protein families[35] and comparison of two Hidden Markov Models[27, 28], to name a few. Several
of our proposed algorithms benefited from these comparison techniques.

8 Conclusion

In this paper, we propose a framework to test quality assurance measures for automaton-based specification miner. Major research issues include model & trace generation and quality assurance metrics & methods.

Several QA metrics are proposed. These metrics cover multiple dimensions of quality in terms of similarity of traces (analogous to recall and precision), structures and probabilities of specification and mined automaton. They should be considered together rather than separately in evaluating a miner. A good miner should have good recall, good precision and be able to retain original structure and probabilities of the original specification. These QA metrics are obtained via language search, automaton alignment and HMM-HMM comparison methods.

Our framework is geared for software engineering purpose for the following reasons: (1) Actual program trace can be mapped to string of alphabets as shown in [1], (2) We have suited the generation of traces and followed known and acceptable coverage criterions rather than simple random walk (Section 3), (3) Generation of models was done following some known software behavior such as the principle of locality (Section 3), and (4) Metrics proposed is directly related to software engineering concerns (Section 4). Precision and recall which is a measure of soundness and completeness ensures usage of generated automata in other software engineering automatic reasoning (e.g. cf. [38, 23]) produces safe results and less false alarms. Structural similarity is useful in aiding software engineers understanding of the resultant automata. Co-emission provides a measure of guarantee on the preservation of probabilities of occurrence of traces. This ensures that "protocol/specification hotspots" are preserved. Its analogy to concepts of hotspots in java compilers [21] might provide glimpse to future use in optimization or other related areas. Measures used in automata research as Minimum Message Length (MML) and relative entropy (cf. [31, 11]) have not been shown to relate well or provide multiple facets of quality of interest in software engineering domain.

Experiments measuring the accuracy, robustness and scalability of sk-strings used by Ammons et al. have been performed. We found that the learner tends to overgeneralize. This resulted in inaccurate inferred specification. The learner is also susceptible to the presence of erroneous traces. The mined specification reduces in precision and structural similarity as we scale up the simulator model under scalability test.

We have compared the quality assurance measures of sk-strings to MMFSM and found that MMFSM can produce more accurate results in the presence of error and increasingly large model. MMFSM should be chosen if error filtering and a more precise inferred automaton is desired. Otherwise, if smaller inferred automata with similar recall is desired, sk-strings should be used instead. In our opinion, the first case is more likely in typical software specification mining tasks.

Another performance issue that has not been addressed so far in our work is the level of user involvement. This estimates the degree of user expertise required and the number of times a user needs to be involved in inferring a specification. Similar issue has also been raised by [32, 26], but an objective measurement of the degree of user involvement is not yet forthcoming.

The framework and metrics developed here do not only provide us a means for quality assurance measurement, they also help discover possible blind spots in the existing research in the field, and provide hints for development of better specification miners to meet the stringent quality assurance requirements. While we acknowledge that some imperfect dynamically learned specification can be useful for software engineering tasks eg. [40], we also believe that improvement in specification miners quality will increase their usefulness even more.

9 Acknowledgments

We would like to thank Anand Raman, Peter Andreae and Jon D. Patrick for letting us use their sk-strings learner in our experiment.

References