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.

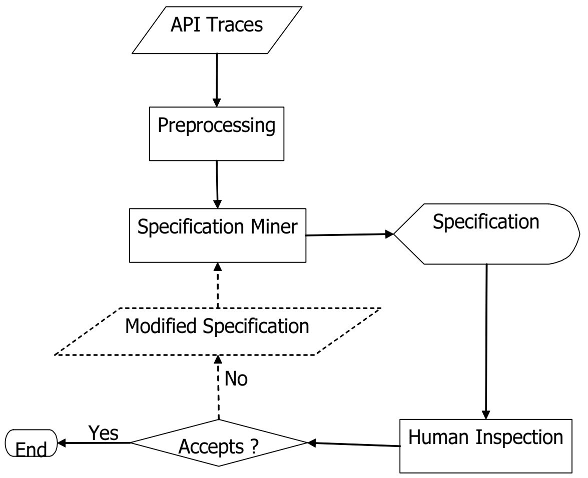


Figure 1. Typical Specification Miner Process

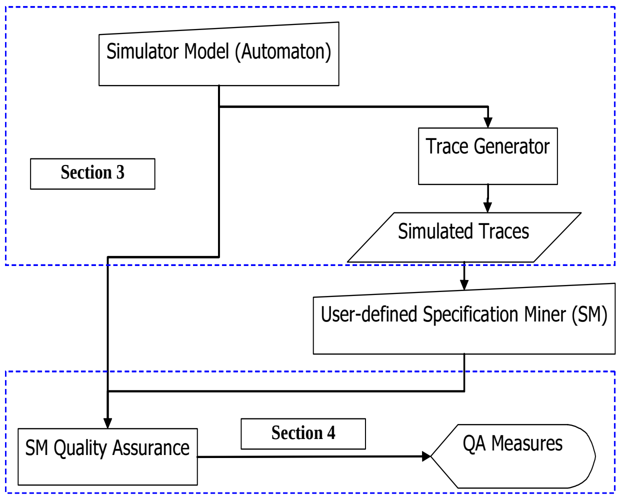


Figure 2. Framework Structure

2.1 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.

2.2 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-

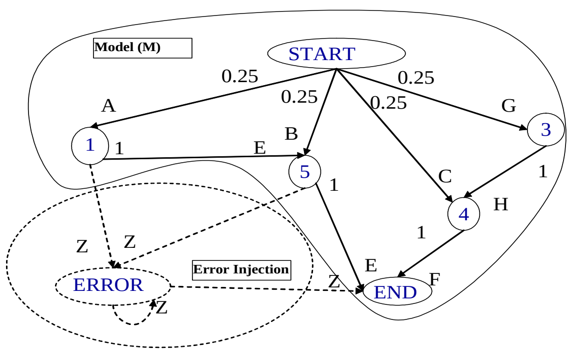


Figure 3. Sample Simulator Model

ure 3. Each node in the automaton represents a program state. There are four types of nodes: start, end, normal and error nodes . Each transition in the automaton represents a viable API method call from that state. For every transition, a probability will be attached to it. The probability attached to a transition indicates how likely the associated method call will be invoked from that source state. It is an invariant of any PFSA under consideration that all transitions emitting from a source, excluding the transitions leading to error nodes (see the following paragraph), must have their probabilities summed up to 1.0.

The model can be *injected with error* by including error nodes and error transitions. Error transitions are modelled using dashed lines in Figure 3. This inclusion of error nodes and error transitions enables generation of erroneous traces; it aids the evaluation of miner's ability to learn in the presence of errors (*i.e.*, robustness). The allocations of error nodes and transitions will characterize the kind of errors allowed. Lastly, we do not assign any probability to error transitions, as we do not intend to micro-manage the generation of error traces.

Furthermore, large (in terms of the number of nodes or the ratio of number of transitions to nodes) models can be inputted to test the scalability of a miner. We automatically generate distinct models having n nodes and maximum of m transitions per state with a common start and end nodes. Transition labels are chosen from a pool of fixed number of labels randomly. Loops are introduced based on the principle of locality where loops between child and parent/ancestor nodes (including self-loop) occur with higher probability than those connecting to distant sibling nodes. The above properties are meant to generate *reasonably* complex models that are more likely to mimic reasonable protocols even in a large system (e.g. business logic of an enterprise application). Hence, with injection of error and variety of model sizes different dimensions of quality assurance can be obtained.

Our model generation algorithm is shown in Figure 4. The algorithm initially create a connected automata structure in the form of a tree until N nodes have been created. Next, depending on loop level and locality level, a set of additional transitions will be introduced creating possibly loops and adding complexity to the model. Loop level and locality level by default are set to 0.4 and 0.8 respectively.

```
Procedure ModelGen
Inputs:
$N$ : Total number of nodes, $> 0$
$M$ : Max number of transitions per node
$Labels$ : Pool of labels
$PLoop$ : Loop level (from 0 to 1)
$PAncestor$ : Locality level : Probability of loop
              to ancestor (including self)
Outputs:
$Model$: Model having N nodes with each node having
         at most M trans with labels from $Labels$.
Method:
Let Root = Create a new node for root node of model
Let NodeList = List of nodes so far, init to {Root}
Let WorkList = List of nodes to process,init to {Root}
// Step 1: Create Connected Automata Structure
while (#NodeList $< N$) do
  Let curNode = A random node from WorkList
  Let transNo = A random no btw 1 to $M$
  Set curNode.maxTrans to transNo
  for (i = 0; i < transNo && #NodeList < N;i++)
      Let newNode = Create a new node
      Add transition from curNode to newNode
        with label randomly pick from $Labels$
      Add newNode to NodeList and WorkList
      if (i == #transNo$-1$)
        WorkList.Remove(curNode)
  end for
end while
// Step 2: Add Loop/Extra Transitions
Let NodeLoopList = Take $PLoop *$ #WorkList
                   nodes randomly frm WorkList.
for each node NLoop in NodeLoopList
     Let transNo = NLoop.maxTrans undefined?
        A random no btw 1 to $M$: NLoopmaxTrans
     while (NLoop has less than transNo trans.)
           Add a loop to NLoop's ancestors with
             prob. $PAncestor$ or to siblings otherwise
     end while
end for
Output  Model with root node set to Root
```

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

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 $i > 0$ 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 $MaxPopN$ – 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

```
Procedure TraceGen2
Inputs:
$M$ : Automaton model ; $EI$ : Error injection
$N$: Cover ; $I$: Maximum number of loop
$MaxPopE$ : Maximum possible error trace population
$MaxPopN$ : Maximum possible normal trace population
$Max$: Maximum trace number
$GE$: Global error injection probability
Outputs:
A multiset of traces
Method:
let $M_{EI}$ = $M \cup EI$
Traverse model $M$
let $Tlist$ = list of all possible paths from $M$ where in
    each path a transition can be traversed at most
    $I$ times capped by $MaxPopN$
let $Plist$ = map between trace in $Tlist$ to its probability
let $Ctr$ = a map from each $t \in Tlist$ to a number
   - initialized to 0
let $Errlist$ = list of all possible error traces bounded by
   $MaxErrPop$ where in each, a transition
   in $M_{EI}$ is traversed at most $I$ times
do {
   Let $rand$ = a random number between 0 to 1
   if ($rand < GE$) {
      Let $TE$ = a trace generated randomly from $Errlist$
      Output $TE$
   }
   else {
      let $trand$ = choose a $t \in Tlist$ according to its
                  probability in $Plist$
      Output $trand$
      $Ctr[trand]$++
   }
} while   ($\exists t \in Tlist$: $Ctr[t] < N$ &
           number of generated traces $\leq Max$)
```

Figure 6. Trace Generation Approach 2

randomly from this list of possible paths according to their probabilities.

4 Specification Miner Quality Assurance

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 Dif_X (Dif_Y) captures the differences between X (Y) and Sim. SS is then a normalized ratio of transitions in similarity and difference automatons:

$$\begin{aligned} \text{SS} &= \tfrac{\sharp Sim.Edges}{w+\sharp Sim.Edges} \\ w &= max(\sharp Dif_X.Edges, \sharp Dif_Y.Edges). \end{aligned}$$

Here, $\sharp X.Edges$ denotes the number of transitions in X.

Probability Similarity. For models that are represented by PFSAs, 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:

$$\begin{aligned} \text{PS}(X, Y) &= \tfrac{2 * P_{CE}(X,Y)}{(P_{CE}(X,X) + P_{CE}(Y,Y))} \\ P_{CE}(X, Y) &= \Sigma_{s \in L(X \cap Y)}(P_X(s) P_Y(s)). \end{aligned}$$

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 automatons that capture structural similarity and difference of X and Y. Based on similarity and difference automatons, we can then calculate the normalized ratio of transitions in similarity and differences automatons. 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 automatons. **(5)** Calculation of SS based on the similarity and difference automatons.

In step 2, *global sequence alignment* is applied to explore all possible alignments of nodes lined up consecutively in the strings $s1$ and $s2$ respectively. Alignment is achieved by assigning a score to each pair of nodes. The score is assigned based on similarities and differences of labels of the outgoing transitions associated with the nodes. The best alignment results are kept in `nodesAlign`[], where each element in the array contains two consecutive subsequences from `s1` and `s2` – 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, ..., x_5, x_{end} and some of those in Y are named y_0, y_1, ..., y_8, y_{end}. A possible pair of alignments can be as follows:

x_0	x_1	x_4	$-$	$-$	x_3	x_2	$-$	x_5	x_{end}
y_0	y_2	y_1	y_4	y_3	y_7	y_8	y_5	y_6	y_{end}

In this example, the “dash” (“-”) represents an added pad. Unification operation performed at Step 3 attempts to discover *segments of similarity* between automatons X and Y by connecting pairs of aligned nodes to form similarity automatons. From the pair of alignments kept in `nodesAlign[i]`, 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 automatons. Probability similarity (PS) can be calculated from P_{CE} as defined

```
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 alignments
Method:
// Step1
Let s1 = Enumeration of all nodes in $X$ using breadth-
         first search of automaton nodes in ascending
         {transition name}++{next node name} order
Let s2 = Enumeration of all nodes in $Y$ using the same
         techniques as that for generating s1
// Step2
nodesAlign[] =
        Global sequence alignment of s1 and s2 (see text)

For each i indexing nodesAlign[] do
  // Step3
  Extract similarity automaton into
      sim[i] by a series of unification        (see text)
  // Step4
  Extract difference automaton into diffx[i] and
  diffy[i] by including transitions in $X$ and $Y$ that
  are not in sim[i]
  // Step5
  Let w =
    max(#diffx[i].Edges,#diffy[i].Edges)
  Let qaMet[i] =
    #sim[i].Edges/(w + #sim[i].Edges)
end for
Output
  nodesAlign[], sim[][], diffx[][], diffy[][],
  qaMet[]
```

Figure 7. Automaton Alignment Algorithm

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 $2 * min(\sharp X.Edges, \sharp Y.Edges)$ 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 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:
Let uX = Enumeration of all nodes in $X$
Let uY = Enumeration of all nodes in $Y$
Let n = number of nodes in $X$
Let m = number of nodes in $Y$
Let dProb[][] = Create table of size n $\times$ m
Let xStart[] = corresponding rows representing start
               nodes of $X$ according to uX
Let yStart[]= corresponding cols representing start
              nodes of $Y$ according to uY
Let xStop[] = corresponding rows representing end
              nodes of $X$ according to uX
Let yStop[] = corresponding cols representing end
              nodes of $Y$ according to uY
Let iterNo = 2*min(#$X$.Edges, #$Y$.Edges)
For each (0 $\leq$ i < #xStart) and (0 $\leq$ 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 $t_h$ and $t_k$ have the same label, then
        Let p($t_h$) = probability of transition $t_h$
        Let p($t_k$) = probability of transition $t_k$
        dSum += dProb[k][h] * p($t_h$) * p($t_k$)
    dProb[i][j] = dSum
$P_{CE}$(X,Y) = Sum of dProb[xStop[i]][yStop[j]]
                for all i and j
Output $P_{CE}$ (X,Y)
```

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

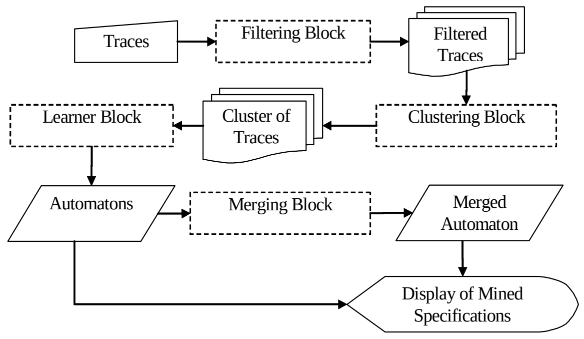


Figure 9. MMSFSM Architecture

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 s% 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 s% 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.

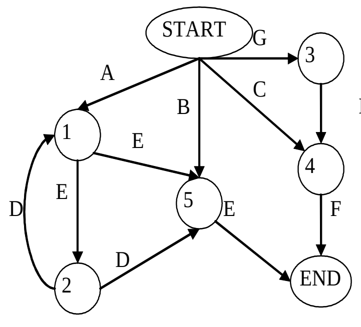
Off-the-shelf sk-strings learner [31] used by Ammons *et al.* and MMFSM was used as the user-defined specification miners for evaluation. Default parameter settings is used for sk-strings both when used stand-alone and within MMFSM. Since MMFSM is meant to improve scalability and robustness, only results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



Origin	Destination	Transition	Probability	Probability[†]
0	1	A	0.25	0.25
0	3	G	0.25	0.25
0	4	C	0.25	0.25
0	5	B	0.25	0.25
1	2	E	0.5	1
1	5	E	0.5	-
2	1	D	0.5	-
2	5	D	0.5	1
3	4	H	1	1
4	END	F	1	1
5	END	E/I[†]	1	1

[†] Used in Experiment 2.

Figure 10. Experiment Simulator Model

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

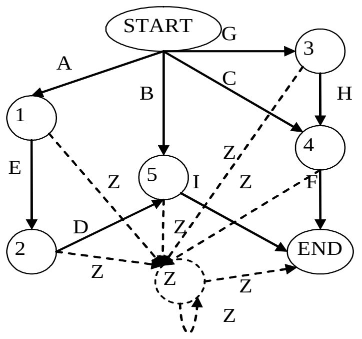


Figure 11. Simulator Model with Error Injected

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) .

Automaton Alignment

The best alignments of nodes is as follows:

START	1	5	4	3	2	End
0	1	-	2	3	4	End0

The similarity automaton Sim is shown in Figure 13. Difference automatons can be extracted by subtracting Sim from X and Y. Structural similarity (SS) can be calculated by normalizing the ratio of the number of transitions in similarity and difference automatons – 0.583 in this case.

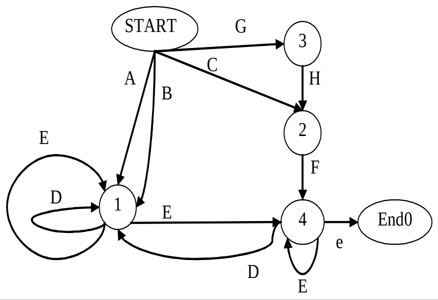
HMM-HMM Comparison Based Technique

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



Origin	Destination	Transition	Probability
0	1	A	0.258
0	1	B	0.245
0	2	C	0.247
0	3	G	0.251
1	1	D	0.138
1	1	E	0.277
1	4	E	0.585
2	4	F	1
3	2	H	1
4	1	D	0.040
4	4	E	0.037
4	END0	ϵ	0.923

Figure 12. sk-strings Mined PFSA and Transition Edges with Their Probabilities

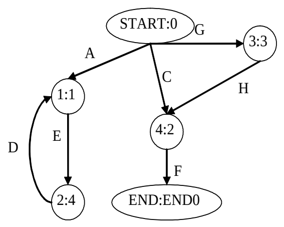


Figure 13. Similarity Automaton

sk-strings learner is still unable to recover the same model. Only 61.017% of the sentences generated by Y using trace generation method 2 are accepted by X. We repeated another experiment by making sure that the model was *without any non-determinism and repeat use of alphabet assigned to transitions*. For this case, the original automaton was able to recover *exactly the same model* with slightly different probability assigned to sentences (PS score 0.944 out of 1). We therefore conclude that sk-strings learner tends to over-generalize. This results in generation of inaccurate inferred model.

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.

E%	TS.XY	TS.YX	SS	PS
4	1.000	0.958	0.693	0.946
8	1.000	0.925	0.597	0.953
10	1.000	0.901	0.549	0.951

Table 1. sk-strings Robustness – Errors

E%	TS.XY	TS.YX	SS	PS
4	1.000	0.996	0.962	0.947
8	1.000	0.990	0.925	0.947
10	1.000	0.982	0.888	0.945

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.

X.N/X.TN	TS.XY	TS.YX	SS	PS
10/4	0.998	0.292	0.192	0.460
20/4	1.000	0.006	0.064	0.022
30/4	1.000	0.003	0.045	0.019
40/4	1.000	0.006	0.022	0.026

Table 3. sk-strings Scalability - Nodes

X.N/X.TN	TS.XY	TS.YX	SS	PS
10/4	0.988	0.588	0.086	0.663
20/4	0.994	0.158	0.034	0.273
30/4	1.000	0.020	0.025	0.088
40/4	1.000	0.016	0.038	0.072

Table 4. MMFSM Scalability - Nodes

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.

X.N/X.TN	TS.XY	TS.YX	SS	PS
10/3	1.000	0.214	0.211	0.341
10/5	0.994	0.206	0.159	0.334
10/7	0.997	0.253	0.226	0.363
10/9	0.998	0.109	0.102	0.221

Table 5. sk-strings Scalability - Max Transitions

X.N/X.TN	TS.XY	TS.YX	SS	PS
10/3	0.998	0.514	0.207	0.574
10/5	0.980	0.489	0.055	0.632
10/7	0.976	0.507	0.049	0.634
10/9	0.976	0.515	0.056	0.625

Table 6. MMFSM 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 acylic 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

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *Proc. of Principles of Programming Languages*, 2002.

[2] G. Ammons, D. Mandelin, R. Bodik, and J. Larus. Debugging temporal specifications with concept analysis. In *Proc. of Programming Language Design and Implementation*, 2003.

[3] D. Angluin. Identifying languages from stochastic examples. *Yale University technical report, YALEU/DCS/RR-614, March 1988*.

[4] T. Arts and L. Fredlund. Trace analysis of erlang program. In *Proceedings of Erlang Workshop*, 2002.

[5] M. R. Betancourt and J. Skolnick. Universal similarity measure for comparing protein structures. *Biopolymers*, 59:305–309, 2001.

[6] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers*, 21:591–597, 1972.

[7] R. Binder. *Testing Object-Oriented Systems Models,Patterns,And Tools*. Addison-Wesley, 2000.

[8] N. Brown, C. Orengo, and W. Taylor. A protein structure comparison methodology. *Computers Chemistry*, 20(3):359–380, 2001.

[9] R. Capilla and J. D. nas. Light-weight product-lines for evolution and maintenance of web sites. In *Proc. of the Seventh European Conference On Software Maintenance And Reengineering*, 2003.

[10] R. Carrasco and J.Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proc. 2nd International Colloquium on Grammatical Inference*, 1994.

[11] R. Carrasco and J. Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *Theoretical Informatics and Applications*, 33:1–19, 1999.

[12] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[13] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *Proc. of The Third Software Product Line Conference*, 2004.

[14] M. El-Ramly, E. Stroulia, and P. Sorenson. Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

[15] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transaction on Software Engineering*, 27(2):99–123, February 2001.

[16] A. Fox. Addressing software dependability with statistical and machine learning techniques. In *Proc. of International Conference of Software Engineering*, 2005. Invited Talk.

[17] E. M. Gold. Language identification in the limit. In *Information and Control*, volume 10, pages 447–474, 1967.

[18] G.Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[19] N. Gupta. Generating test data for dynamically discovering likely program invariants. In *Proc. of the workshop on dynamic analysis*, 2003.

[20] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.

[21] The java hotspot performance engine architecture. white paper available at http://java.sun.com/products/hotspot/whitepaper.html, apr. 1999.

[22] J.Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proceedings of the European Conference of Object Oriented Programming*, 2003.

[23] P. S. K. Marriott and M.Sulzmann. Resource usage verification. In *Proceedings of the Asian Symposium Programming Languages and Systems*, 2003.

[24] M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. Schapire, and L. Sellie. On the learnability of discrete distributions. In *Proc. of the Twenty-sixth ACM Symposium on Theory of Computing*, 1994.

[25] E. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Mining and Knowledge Discovery*, 7:349–371, 2003.

[26] E. Keogh, S. Lonardi, and C. Ratanamahatana. Towards parameter-free data mining. *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD'04)*, 2004.

[27] R. Lyngsø, C. Pedersen, and H. Nielsen. Measures of hidden markov model. *BRICS Report Series*, 1999.

[28] R. Lyngsø, C. Pedersen, and H. Nielsen. Metrics and similarity measures for hidden markov models. In *Proc. of the 14th National Conference on Artificial Intelligence*, 1999.

[29] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. of the 2002 International Symposium on Software Testing and Analysis*, 2002.

[30] S. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Enginering*, 14(6):868–874, 1988.

[31] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *Proc. of the workshop on automata induction, grammatical inference and language acquisition*, 1997.

[32] O. Raz, P. Koopman, and M.Shaw. Enabling automatic adaptation in systems with under-specified elements. In *Proceedings of Workshop on Self-Managing System*, 2002.

[33] O. Raz, P. Koopman, and M.Shaw. Semantic anomaly detection in online data sources. In *Proc. of International Conference on Software Engineering*, 2002.

[34] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the International Conference on Software Engineering*, 2001.

[35] J. S̀oding. Protein homology detection by hmm-hmm comparison. *Bioinformatics*, 21(7):951–960, 2003.

[36] M. Tompa. Lecture notes on biological sequence analysis. *Technical Report 2000-06-01 University of Washington*, 2000.

[37] C. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[38] D. Wagner and D.Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[39] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object oriented component interfaces. In *Proc. of the International Symposium on Software Testing and Analysis*, 2002.

[40] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.