QUARK: Empirical Assessment of Automaton-based Specification Miners

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

Abstract

Software is 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. There has been much work on reverse engineering or 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 QUARK(QUality Assurance framewoRK) that enables assessments of the performance of a specification miner in generating temporal specification of software through traces recorded from its API interaction. QUARK requires the temporal specification produced by the miner to be expressed as an automaton. It accepts a user-defined simulator automaton and a specification miner. It produces quality assurance measures on the specification generated by the miner. Extensive experiments on 3 specification miners have been performed to demonstrate the usefulness of our proposed framework.

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 [7] 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.* [10]).

Recently, there has been a surge in software engineering research to adopt machine learning and statistical approaches to address these problems. One active area is specification discovery [8, 19, 22, 1], where software specification is reverse-engineered from program traces. In [11], Fox illuminated 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.* coined the term *specification mining* as a machine learning approach to discover program specification by analyzing program execution traces [1]. Under the assumption that programs being mined must "reveal strong hints of correct protocols" during their execution, Ammons *et al.* demonstrate that correct specifications can be obtained through their technique. Specifically, their technique focuses on mining of specification which reflects temporal and data dependency relations (*i.e.*, *temporal specification*) of a program through traces of its API-client interaction. The specification miner discovered API-client interaction protocol model, 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 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 literature on software specification validation and program analysis. During our assessment, we generate program traces from a chosen specification, use the traces to mine a specification, and then compare the mined specification against the original specification. A good specification miner should infer a specification that matches the original specification as accurately as possible, if the set of traces generated is a good representation (sample) of the original specification. Our measurement of *accuracy* is adapted from the measurements of *recall* and *precision* of Nimmer *et al.* in evaluating Daikon. Nimmer *et al.* also relates these measurements to the concept of soundness and completeness used in program analysis community [20]. On the other hand, we do not advocate measuring *compactness* of mined specification against the *training set* of traces as often found in machine-learning literature (*aka.*, Minimum Message Length [21, 4, 15]), as we believe *accuracy* is a more relevant issue than *compactness* for software specification mining.

An additional advantage of having these objective assessments of specification miner is that they not only define the quality of specification miners in different dimensions, but also highlight areas for improvement, and 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. Our framework (QUARK) requires any specification miner under assessment to exhibit 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 finite-state automaton.*

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

The original automaton can be either probabilistic or not (PFSA/ FSA). In fact, an FSA is a special form of PFSA with probability dropped. Representing specifications as *Probabilistic FSAs* (PFSAs) instead of FSAs, however, has the following benefit: *Probabilities attached to a protocol specification enable more control over the trace-generation process so that the collection of traces generated mimics certain characteristics of the traces that can be collected from actual API interactions.* For example, sub-protocols within a protocol specification may appear more frequently than others in the actual interaction with API interface – analogous to the idea of hotspot found in program execution [24]. 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.

In addition, it has also been proven that perfect learning of an FSA from positive examples is not decidable [1, 12], whereas perfect learning of a PFSA from examples is decidable (*cf.* [9, 3]) though inefficient (*cf.* [16]). This theoretical finding has prompted Ammons *et al.* to use PFSA as an intermediate step to the learning of an FSA.

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

Furthermore, by varying the percentage of expected error and the size of the original automaton, QUARK enables the respective assessments of robustness and scalability of the miners.

We have built a prototype of QUARK, and used it to assess some existing specification miners. In this paper, we describe our comprehensive experiments on three specification miners. These experiments include mining of several real-world API-interaction specifications obtained from (1) programs using XLib and XToolkit intrinsic libraries for X11 windowing system [1], (2) IBM® WebSphere® Commerce code [27], and (3) a simple CVS protocol built on top of Jakarta Commons Net [25].

The outline of the paper is as follows: In section 2, a typical specification mining process is presented and the structure of QUARK is outlined. Sections 3 and 4 describe our solutions to two major issues related to quality assurance measurement: model-and-trace generation and the metrics and techniques for quality assurance. Section 5 briefly describes specification miners used in our experiments. Sections 6 and 7 describe our experiments and results. We discuss related work and conclude in Section 8.

2 Framework Structure

Typically, a miner's input is in the form of API interaction traces, where each trace represents a sequence of

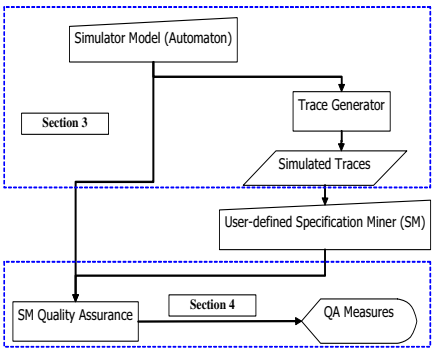


Figure 1. Framework Structure

method calls (with or without argument). Preprocessing is usually performed on these traces to turn each of them into an abstract trace. We omit the detail of instrumentation, collection and abstraction of program traces, as these have been documented in the literature; *e.g.*, [1], [26], *etc.* Specification miner then learns from these traces to produce a specification. The specification can be expressed in various forms: automaton, algebraic equations, Hoare-style equation of pre and post-condition, *etc.* Human judgment is often employed at this stage to assess the performance of the miner. Some systems, such as [1], in addition permits mined specification to be modified manually before returning.

Some existing systems, such as Daikon, are assessed by measuring their accuracy in recalling correct information (invariants) and in reducing the generation of incorrect information [20]. However, they fall short in 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.

QUARK aims to address all the above quality assurance concerns for assessment of automaton-based specification miners. 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 a miner's performance in recovering the original model from three dimensions: its accuracy, robustness and scalability.

The structure of QUARK is shown in Figure 1. 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 sub-system* to generate various quality assurance metrics.

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

3 Simulator Model & Trace Generation

QUARK admits two closely-related simulator models: FSA and PFSA. In both cases, it accepts both deterministic and non-deterministic models. Since PFSA is technically more complex to handle than FSA, we focus our discussion on PFSA and its associated trace-generation method. At the end of this section, we show how our method can be adapted to handle FSA.

3.1 Probabilistic Model

Figure 2 depicts an example of error-injected simulator model. Ignoring the dotted nodes and dashed edges, the remaining model is in the form of a probabilistic finite state automaton (PFSA). Each node in a PFSA represents a program state. There are 3 types of nodes: start, end and normal nodes. Each transition in the automaton denotes an abstract representation of a viable API method call from that state. Every transition is attached with a probability, indicating 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) must have their probabilities summed up to 1.0.

Error Injection A PFSA model can be *injected with error* by including error nodes and error transitions, shown as dotted nodes and dashed edges respectively in Figure 2. This inclusion enables generation of erroneous traces, and aids the evaluation of miner's ability to learn in the presence of error (*i.e.* robustness). The allocations of error nodes and transitions characterize the kind of errors allowed. Probabilities are not assigned to error transitions, as we do not intend to micro-manage the generation of erroneous traces. We will describe generation of erroneous traces in Section 3.2.

Model Size and Model Generation In addition to subjecting miners to tests with real-world specifications, we also devise ways to generate synthetic models. This allows us to perform controlled experiments on miners' quality.

To test a miner's scalability, we control the size of a simulator model by varying the number of nodes it has and the maximum number of transitions a node can emit. We automatically generate distinct models having n nodes and a maximum of m transitions per state with a common start and end nodes. Transition labels are chosen randomly, with repetition, from a pool of fixed number of labels. We first build a tree from a pre-determined number of nodes. Next,

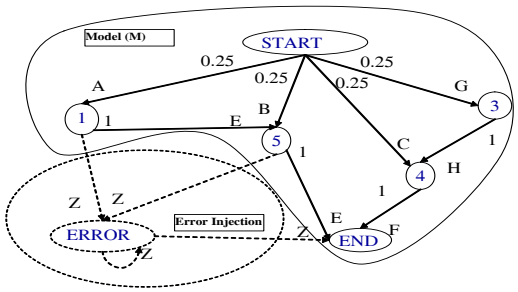


Figure 2. Sample Simulator Model

to mimic the behaviour of typical API-interaction, we introduce loops into the tree based on an idea similar to the principle of 'locality of reference'. This well-known principle states that a program tends to reuse data and instructions it has used recently [23]. Adapting from this principle, a method will more likely be invoked again if it has just been called before. Hence, loops between child and parent/ancestor nodes, including self-loop, are introduced with higher probability than those connecting to distant sibling nodes. Although we have not rigorously verified the applicability of this principle, the three real-world specifications shown in Section 6 are found to adhere to this principle.

Lastly, probabilities are assigned equally to transitions from the same source node. Due to lack of space, we refer readers to [17] for the algorithm detail of the synthetic model generator.

3.2 Trace Generation

Actual program trace can be mapped to string of alphabets, as shown by Ammons *et al.*, through a 'standardization' process, in which an alphabet (corresponds to a transition label in the simulator model) represents a particular method call [1]. Two types of traces are generated: normal and erroneous traces. A *normal 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. An *erroneous trace* is one that includes an error transition.

Since normal traces are generated from a PFSA, we can determine the probability of a trace by multiplying together the probability of its constituents.

Given an input model, the algorithm for trace generation is described below. Basically, it performs a *stratified random walk* over the input model, guided by the probability of the PFSA's transitions. Consequently, it ensures 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.)

This algorithm, called **TraceGen**, is akin to the "code and branch coverage" criterion used in generating program test cases [6]. Given a PFSA M, a cover N, and a maximum trace number Max , **TraceGen** generates a multiset of traces T possessing the following asymptotic property:

Property 1 *For any $N > 0$, and for a sufficiently large number Max, every transitions in the PFSA M occurs at least N times in the traces multiset T of size at most Max.*

This property ensures that all transitions in M have the opportunity to be used for trace generation. This is the *coverage criterion* used in our experiments. The algorithm detail is depicted in Figure 3.

Procedure TraceGen
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 3. Trace Generation Algorithm

In Figure 3, 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 an 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.

Erroneous Traces Generation. The percentage of erroneous traces generated are controlled at a global level by a probability GE. Before a trace is generated to fill up the trace set, the algorithm checks if erroneous trace needs to be generated. If so, such a trace is produced by choosing one randomly from previously generated pool of erroneous traces ($Errlist$ - see Figure 3).

Non-Probabilistic Model. An FSA model can be easily obtained from a PFSA simulator model by dropping the probability associated with each transition in the model. The *major technical difference* between using FSA and PFSA simulator models is trace generation. In FSA, a *standard random walk* is performed, rather than *stratified random walk*. For the algorithm in Figure 3, this difference occurs at the program point (*): When FSA is used, a normal trace is generated by starting from start node of the model and randomly choosing an outgoing transition to reach the next node, until the end node is reached. Here, all outgoing transitions from a node have *equal chance* to be chosen.

4 Specification Miner Quality Assurance

The quality of a specification miner is measured along three dimensions: accuracy, robustness and scalability.

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*. Erroneous traces usually constitute a small proportion of the entire collection of traces, and a robust miner should be able to filter erroneous traces in building mined models.

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. Central to our assessments is a thorough treatment of accuracy. In the rest of the section, we shall provide a detailed account of metrics and techniques employed in measuring accuracy.

4.1 Trace Similarity

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*. For clarity sake, we denote a simulator model by X and a mined model by Y. We use the term "sentence" and "trace" interchangeably.

In assessing accuracy, we adopt two metrics to measure the similarity between X and Y in terms of their *generated traces* (or sentences). First, the percentage of sentences *generated by X* that are *accepted by Y* represents the amount of correct information that can be recollected by the mined model. This measurement is known as *recall* in information retrieval literature (*cf.* [13]). Second, the percentage of sentences *generated by Y* that are *accepted by X* represents the amount of correct information that can be produced by the mined model. This is known as *precision* (*cf.* [13]).

The notions of *recall* and *precision* are also used by Nimmer *et al* to evaluate Daikon. Nimmer *et al.* further relate them to measures of completeness and soundness, respectively [20].

To perform trace similarity measurement, we employ an *automaton language search technique*. This basically generates two sets of samples of traces from X and Y, respectively, and calculates the percentage of traces generated by X that are accepted by Y, and vice versa. The trace sample generated from X will be different from the set of traces used in training the miner. Separating the training set from the test set enables the detection of any "overfitting" done by the miner; *ie.*, the miner learns the training set so closely that it does not generalize well to original model [14].

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 **TraceGen** procedure in Figure 3 to help in trace generation.

4.2 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 third 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.

Co-emission has been used in measuring probability similarity between two Hidden Markov Models [18]. Let $L(M)$ represent the language recognized by the automaton M, the co-emission is defined by the following formula:

$$P_{CE}(X,Y) \quad = \quad \Sigma_{s \in L(X \cap Y)}(P_X(s)P_Y(s)).$$

Here, $P_{CE}(X,Y)$ determines 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.

The *probability similarity* between X and Y, denoted by PS, can then be defined as follows [18]:

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

This provides an unbiased and normalized probability similarity measurement of the two models. In practice, this computation is realized by a *HMM-HMM comparison-based*

technique. This technique has been adapted from the work of Lyngsø *et al.* [18]. Due to lack of space, we refer readers to [17] for detailed discussion of this technique.

5 Specification Miners Used

In this section, the three specification miners used in our experiments will be briefly described. They are: (1) *k-tails FSA learner*, (2) *sk-strings PFSA learner* and (3) our own miner (Specification Mining Architecture with Trace Filtering and Clustering – *SMArTIC*) which produces PFSA.

k-tail algorithm is a well-known *heuristic* algorithm proposed by Biermann and Feldman [5] to learn automata from positive samples. It has been adapted/modified by various researchers to perform specification mining tasks [8, 22, 19]. From a training set of positive samples, the algorithm first builds a prefix tree acceptor. Informally, a prefix tree acceptor (PTA) is an automata in the form of a tree where there is one node for every common prefix and each leaf is a final state. Given a PTA, a node q, a set of alphabet Σ, a set of final states (the leaves of PTA) F_c, and an extended transition function δ^* , the set of k-tails associated with the node q is given by $\{s | s \in \Sigma^*, |s| \leq k \wedge \delta^*(q, s) \cap F_c \neq \emptyset\}$. Two nodes form this PTA are then merged if their respective k-tails are indistinguishable.

sk-strings algorithm is an extension of k-tails heuristic for stochastic automata. It has been used by Ammons *et al.* in [1]. Similar to k-tails, sk-strings algorithm also builds a prefix tree acceptor from traces. The difference lies in the criteria for merging of nodes and for incorporation of probability estimation. Two nodes are merged if they are indistinguishable with respect to the *top s% most probable strings* (instead of *tails*) of length *exactly k* (or less if an end node is encountered before reaching length k) that can be generated starting from them.

The default parameters of sk-strings [21] as implemented by Raman *et al.* are: $s\%$=50% and k=1. Also, by default, an AND variant of the algorithm is used. Unless otherwise stated, these defaults are used in the experiments (k=3 is also used in some of our experiments).

In [1], Ammons *et al.* discussed *coring* method as a post-processing step to remove erroneous transitions from the mined automaton. Briefly, identification of erroneous transitions is determined by a notion of *heat*. The heat between a source node and a sink node is the probability that the sink is reached from the source in any amount of steps. A low *heat* transition is likely to be erroneous and will be pruned. In this paper, we refer to sk-strings with *coring* as sk-coring.

Specification Mining Architecture with Trace Filtering and Clustering (SMArTIC) comprises 4 major blocks: Clustering, Filtering, Learning and Merging, as shown in Figure 4.

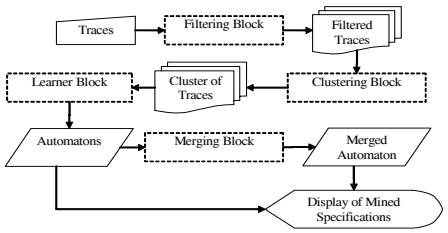


Figure 4. SMArTIC Architecture

Traces deviating from common trace population rules are deemed to be erroneous, and are removed. Contrary to sk-coring, removal of error is done prior to automata learning and not after. The resultant filtered traces are then separated into multiple clusters whose sizes are determined automatically. By clustering common traces together, the learner is expected to learn better since this restricts the effect of over-generalization to within a cluster. Each cluster can be considered as an independent sub-protocols; each will be fed separately to a specification miner. Among others, we use sk-strings learner for convenient sake. The multiple mined automata are then merged, with no further generalization, to obtain the final automaton.

6 Experiments

Three sets of experiments were conducted to show the usefulness of QUARK in evaluating the performance of the three specification miners described earlier. These experiments aims to measure the accuracy of these miners in discovering various real-world specifications.

Material Simulator models used in these experiments are specifications from (1) programs using XLib and XToolkit intrinsic libraries for X11 windowing system [1], (2) IBM® WebSphere® Business Integration processes from WebSphere® Commerce [27] (3) Simple CVS (Concurrent Versions System) protocol built on top of Jakarta Commons Net [25]. These simulator models are shown in Figure 5, 6 and 7, and are referred to as *x11*, *ws* and *cvs* models respectively. Probabilities are *distributed equally* to transitions from the same source node (not shown in figures).

For each model, 100 experiments were run for each learner with the k parameter set to 1 and then to 3. A total of 1800 experiments were performed. For each experiment, a multi-set of traces was generated from the model using **TraceGen** (Figure 3) with parameters N, I and Max set to 10, 10 and 10,000 respectively. No error was introduced to the models.

In analyzing the results, any two results differing in absolute value by less than 1%(0.01) are considered equivalent, as the difference is deemed insignificant.

X11 Windowing Toolkit In [1], Ammons *et al.* described

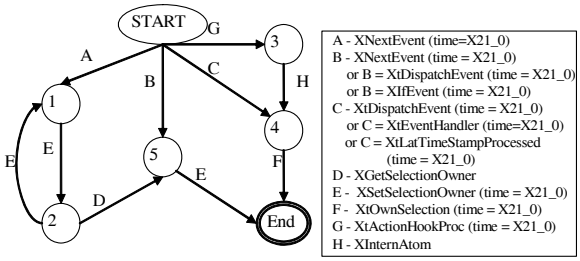


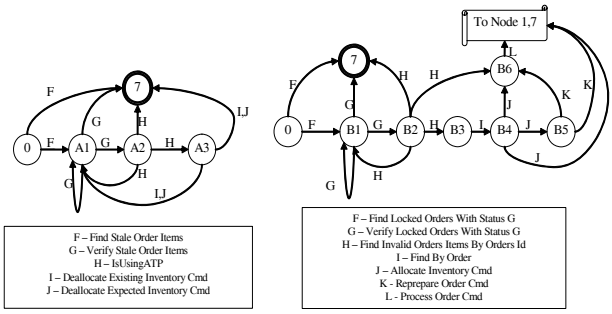
Figure 5. X11 Windowing Toolkit

the mining of a specification, shown in Figure 5, from several programs using XLib and XToolkit intrinsic libraries for X11 windowing system.

In our experiments, the mining results obtained by the three learners are shown in the table below. k-len corresponds to the k parameter of sk-strings (used by both sk-strings and SMArTIC) and k-tails algorithms. A default value of 50% for s was used. The columns Recall, Precs. and PS are the QA metrics defined in Section 4.

Learner	k-len = 1			k-len = 3		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	0.998	0.313	N/A
sk-strings	1.000	0.654	0.692	0.998	0.883	0.758
SMArTIC	1.000	0.820	0.910	0.998	0.987	0.956

Analysis The results show that: (1) k-tails did not learn well at k-len = 1, while sk-strings learnt reasonably well. (2) With bigger k-len, all miners produced more precise automata. (3) sk-strings produced more precise automaton than k-tails, and SMArTIC improved upon sk-strings in both its precision and probability similarities.



(a) **(b)**

Figure 6. WebSphere® Commerce Processes

WebSphere® Commerce. In [27], Zou *et al.* statically extracted workflows describing IBM® WebSphere® Business Integration business process from the IBM® WebSphere® Commerce code. They presented two workflows, in the form of automata, which correspond to (1) the release of expired allocations and (2) the processing of

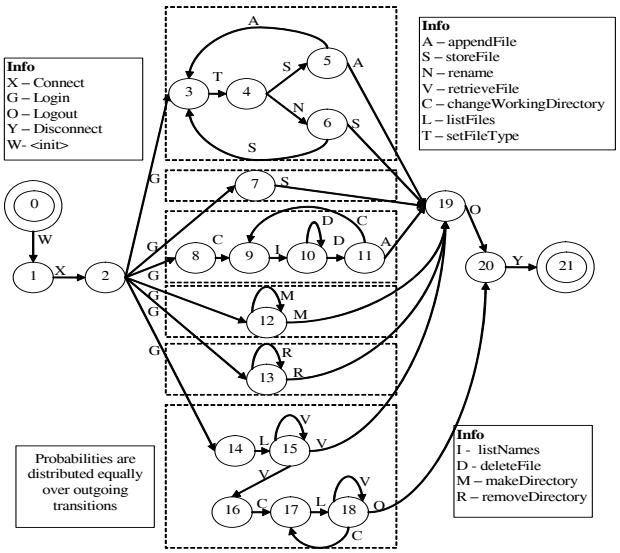


Figure 7. CVS Protocol

backorders. These are shown in Figures 6(a) and (b), respectively. We combine the two automatons into a simulator model by joining their start and end nodes. Note that, different from *x11* and *cvs* models, this model has more transitions per nodes and more loops (*ie.* it is more "bushy"). The experiment results are tabulated in the following table .

Learner	k-len = 1			k-len = 3		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	0.998	0.597	N/A
sk-strings	1.000	0.536	0.785	1.000	0.538	0.785
SMArTIC	1.000	0.779	0.780	1.000	0.753	0.783

Analysis The results show that: (1) k-tails did not learn well at k-len = 1 as compared with sk-strings. (2) Increasing the value of k-len did not improve the performance of sk-strings, and even caused a slight degradation in the performance of SMArTIC. (3) sk-strings performed worse than k-tails for k-len=3, wheras SMArTIC improved upon sk-strings in its precision, and had better results than k-tails.

CVS on Jakarta Commons Net Jakarta Commons Net [25] is a set of reusable open source java code implementing the clients of many commonly used network protocols. We built a simple CVS (Concurrent Versions System) client on top of the FTP library provided by Jakarta Commons Net.

There are six common FTP interaction scenarios in our CVS implementation: Initialization, multiple-file upload, download, and deletion, multiple-directory creation and deletion. All scenarios begin by connecting and logging-in to the FTP server. They end by logging-off and disconnecting from the FTP server. The client side only maintains a record of files backed-up in the FTP server.

All these scenarios are depicted in the automata shown in Figure 7. The dashed boxes, from top to bottom, represent

upload files, initialization, delete files, make directories, remove directories and download files scenario, respectively.

Compared with *x11* and *ws* models, this model has the most number of nodes, but it remains to be less 'bushy'. The experiments results are tabulated below.

Learner	k-len = 1			k-len = 3		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	1.000	0.000	N/A
sk-strings	1.000	0.226	0.509	0.999	0.017	0.030
SMArTIC	0.986	0.487	0.669	0.973	0.503	0.523

Analysis The results show that: (1) k-tails did not learn well at k-len = 1 *and* k-len = 3. (2) At k-len = 3, the performance of sk-strings was degraded, whereas that of SMArTIC improved sightly. (3) sk-strings performed better than k-tails, while SMArTIC improved upon sk-strings in both precision and probability similarities.

7 Robustness and Scalability

Two sets of experiments were conducted to evaluate the robustness and scalability of the three miners. In total, 2400 robustness experiments were run to cover three error-injection levels, four learners and two k-len values. Also, 2400 scalability experiments were run to cover eight different pairs of node-numbers and maximum number of transitions per node, three learners and two k-len values.800 different models were used in the scalability experiments (*ie.* experiments with the same settings but for different learners shared the same model and trace multi-set).

Material In the first set of experiments, we evaluated the learners' robustness. We used similar model of X11 Windowing Toolkit (shown in Figure 5). However, the model was modified so that it was *without any non-determinism nor repeated use of alphabet assigned to transitions*. This is meant to produce a base model that can be learned (almost) perfectly by all miners. Error nodes and transitions were then injected to the automaton to conduct the robustness tests. The model used with injection of errors (transitions labelled as Z) is shown in the Figure 8.

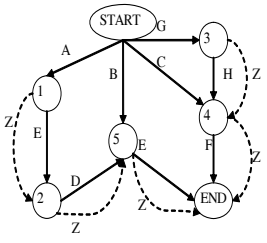


Figure 8. Robustness Simulator Model

We expect specification miner to be able to filter error. We compared the inferred automaton with the simulator model shown in Figure 8 without error nodes and transitions and recorded their similarity metrics. We generated traces using **TraceGen** (Figure 3) with parameters N, I and Max set to 10, 10 and 10,000 respectively. Error was injected at four, eight and ten percentages to the set of generated traces. In each case, we ran 100 experiments and recorded the average performance.

In addition to testing the three learners, we also tested sk-coring (combining sk-strings and coring method). In order to analyze the effect of the filtering block of SMArTIC (which is meant to ensure robustness), we only enabled this block. ('Full' SMArTIC was used for mining specifications extracted from real software in Section 6.)

In the second set of experiments, we evaluated the learners' scalability. Two sub-experiments were conducted, each with a different independent variable. In the first sub-experiment, we varied the number of nodes (by 15, 20, 25, and 30) in the model and maintained the number of outgoing transitions per node to *at most* four (we refer to it as *nodes* experiment). In the second sub-experiment, we varied the maximum number of outgoing transitions per node (by 3,5,7,9) and maintained the number of nodes at 10 (we refer to it as *trans* experiment). For each case, we performed 50 experiments and recorded their average performance.

We generated traces using **TraceGen** with parameters N, I and Max set to 10, 10 and 10,000 respectively. No error was injected to the system. Since we imposed a cap of Max traces, there might be a concern that training trace-set does not satisfy the coverage criterion by *merely* generating up to Max traces. Fortunately, this did not happen that often, as only 18 out of 2400 experiments reached the cap; for all other experiments, the coverage criterion was met without the need to generate Max traces.

The three usual learners were tested. To analyze the effect of clustering block of SMArTIC (which is meant to ensure scalability), only this block is enabled.

Robustness Experiment Results These are tabulated in the following table . Column E% indicates the percentiles of erroneous traces.

Info		k-len = 1			k-len = 3		
E%	Learner	Recall	Precs.	PS	Recall	Precs.	PS
4%	k-tails	1.000	0.000	N/A	1.000	0.763	N/A
	sk-strings	1.000	0.944	0.947	1.000	0.949	0.948
	sk-coring	0.756	0.956	0.831	0.764	0.965	0.838
	SMArTIC	1.000	0.992	0.945	1.000	0.994	0.947
8%	k-tails	1.000	0.000	N/A	1.000	0.645	N/A
	sk-strings	1.000	0.892	0.944	1.000	0.899	0.944
	sk-coring	0.781	0.903	0.828	0.795	0.916	0.835
	SMArTIC	1.000	0.989	0.945	1.000	0.990	0.945
10%	k-tails	1.000	0.000	N/A	1.000	0.621	N/A
	sk-strings	1.000	0.864	0.935	1.000	0.872	0.933
	sk-coring	0.754	0.873	0.800	0.761	0.900	0.803
	SMArTIC	1.000	0.980	0.936	1.000	0.982	0.935

Analysis The presence of error affected miners' precision. We rank the learners' precisions in decreasing order wrt the degrees of their susceptibility to errors as follows: k-tails,

sk-strings, sk-coring and SMArTIC. Also, increasing k-len value *did not* significantly reduce the susceptibility to error.

For sk-coring and sk-strings, losses in precision were about the same as the percentages of error injected. For k-tails however, the losses of precision were much larger. Although sk-coring removed error and improved precision, the ability to recall was adversely affected. On the other hand, we note that SMArTIC was only slightly affected by the increase in the number of erroneous traces.

Scalability Experiment Results The results of our two sub-experiments are shown below. Column "N/TN" corresponds to the number of nodes and the maximum number of transitions per node in the simulator models.

Info		k-len = 1			k-len = 3		
N/TN	Learner	Recall	Precs.	PS	Recall	Precs.	PS
15/4	k-tails	1.000	0.002	N/A	0.999	0.195	N/A
	sk-strings	1.000	0.094	0.152	0.997	0.296	0.344
	SMArTIC	0.996	0.346	0.466	0.982	0.773	0.757
20/4	k-tails	1.000	0.004	N/A	0.997	0.138	N/A
	sk-strings	1.000	0.025	0.059	0.997	0.338	0.371
	SMArTIC	0.996	0.175	0.287	0.985	0.655	0.661
25/4	k-tails	1.000	0.007	N/A	0.998	0.089	N/A
	sk-strings	1.000	0.008	0.029	0.997	0.123	0.197
	SMArTIC	0.998	0.106	0.195	0.988	0.490	0.519
30/4	k-tails	1.000	0.008	N/A	1.000	0.064	N/A
	sk-strings	1.000	0.007	0.031	0.999	0.079	0.105
	SMArTIC	0.999	0.077	0.199	0.991	0.353	0.360

Info		k-len = 1			k-len = 3		
N/TN	Learner	Recall	Precs.	PS	Recall	Precs.	PS
10/3	k-tails	1.000	0.002	N/A	0.998	0.201	N/A
	sk-strings	1.000	0.165	0.283	0.992	0.928	0.913
	SMArTIC	0.991	0.416	0.536	0.977	0.984	0.963
10/5	k-tails	1.000	0.004	N/A	0.980	0.494	N/A
	sk-strings	0.997	0.294	0.375	0.976	0.626	0.614
	SMArTIC	0.979	0.538	0.667	0.957	0.860	0.819
10/7	k-tails	1.000	0.007	N/A	0.963	0.446	N/A
	sk-strings	0.999	0.142	0.203	0.960	0.420	0.173
	SMArTIC	0.986	0.453	0.553	0.939	0.753	0.717
10/9	k-tails	0.997	0.008	N/A	0.934	0.467	N/A
	sk-strings	0.999	0.082	0.141	0.979	0.338	0.339
	SMArTIC	0.976	0.432	0.508	0.934	0.759	0.696

Analysis For all learners, their recalls were always greater than 90%. The average recalls for k-len = 1 and 3 were 99.6% and 96.6% respectively. In each experiment setting, recalls of different learners only differs by less than 5%. However, the precision results were less glossy. Even for k-len = 3, there were cases where precisions were less than 10% (*see* k-len=3;N=30;TN=4). The average precision for k-len = 1 and 3 are 14.2% and 45.3% respectively.

sk-strings' precision is almost always equivalent to or better than k-tails', except for very "bushy" automaton (*see* k-len=3;N/TN=10/9). Similar results were reported in the *ws* experiment described in Section 6. k-tails did not perform well with k-len=1 (precision $< 1\%$). Increasing the "bushiness" of models – by increasing TN from 1 to 9 for 10-node automatons – improved the relative performance of k-tails over sk-strings.

For all cases, SMArTIC had better overall performance in terms of precision over both k-tails and sk-strings, and probability similarity (PS) over sk-strings. The differences were significant, especially for large number of nodes (*see* k-len=3;N/TN=30/4). Its ability to recall is only slightly less than those of the other learners, with their differences capped at 4.5% (*see* k-len=3;N/TN=30/4), and averaged at 1.7%.

8 Related Work and Conclusion

In this paper, QUARK, a framework to empirically assess quality of automaton-based specification miner is proposed. Our assessment of specification miners is guided by the conviction that: A good miner should have good recall, good precision and be able to retain probability distribution of the original specification (for PFSA learner). In addition, it should remain robust in the presence of error, and scalable in learning from traces generated from large automata.

There have been numerous work in the research of automaton-based specification mining [1, 8, 22, 19]. Experiments provided in these works have given guarantees to the quality of the proposed miners. These guarantees can be *further strengthened* by our comprehensive quality assurance metrics and simulation-based framework.

Nimmer *et al.* provide a precision- and recall-based quality measures for Daikon - which generates Hoare-style equation of pre and post conditions [20]. Lyngsø *et al.* provide a similarity measures for probability distribution of Hidden Markov Model [18]. In this paper, we adapt these metrics to our framework as a means for measuring the accuracy of automata generated by specification miners.

To demonstrate the effectiveness of QUARK in assessing specification miners, we use it to assess three types of automaton-based specification miners: (1) *k-tails FSA learner* (2) *sk-strings PFSA learner* and its variant (*sk-coring*) and (3) our own miner (Specification Mining Architecture with Trace Filtering and Clustering – SMArTIC).

Experiments using real-world specifications from X11 Windowing Toolkit, WebSphere® Commerce and CVS were performed. Results show that for *x11* and *cvs* models, sk-strings performed better than k-tails. For *cvs* model, k-tails did not learn well even when k is set to 3. However, for *ws* model, k-tails performed slightly better than sk-strings. It is noted that for all cases SMArTIC had better performance.

Simulated experiments measuring robustness and scalability of the miners were also performed. The results indicate that specification miners typically have good recall ability but poor precision in the presence of error, resulting in inaccurate inferred specification. Our preliminary work in addressing this problem leads to the creation of SMArTIC. In the scalability experiments, increasing the number

of nodes in simulator models can reduce recall ability; increasing the number of transitions per node in simulator models leads to narrowing in the performance gap between k-tails and sk-strings. For very "bushy" simulator models, k-tails perform better than sk-strings. Again, it was noted that for all cases SMArTIC had better results.

In summary, QUARK is specially designed to assess automaton-based *specification* miners rather than generic automaton miners, since: **(1)** Generated traces are viewed as abstract representation of actual program traces; **(2)** trace generation conforms to 'code and branch coverage'-based criterion; **(3)** various models extracted from real software have been used; **(4)** synthetic models are generated following the principle of locality of reference; and **(5)** metrics proposed are directly related to software engineering concerns.

The framework and metrics developed here do not only provide us a means for quality assurance measurement. They also provide hints for development of better specification miners to meet the stringent quality assurance requirements. While we acknowledge the usefulness of producing imperfect learned specification in meeting certain software engineering tasks, we also believe that improvement in specification miners' quality will greatly enhance their usefulness.

Acknowledgments We would like to thank Anand Raman, Peter Andreae and Jon D. Patrick for letting us use the implementations of sk-strings and k-tail algorithms in our experiments. We would also like to thank Glenn Ammons and Rastislav Bodik for sharing the detail of their coring algorithm.

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 tech. report, YALEU/DCS/RR-614*, 1988.

[4] A.Raman, P.Andreae, and J.D.Patrick. A beam search algorithm for pfsa inference. *Pattern Analysis and Applications*, 1998.

[5] 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.

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

[7] R. Capilla and J. C. Dueñas. Light-weight product-lines for evolution and maintenance of web sites. In *Proc. of the Euro. Conf. On Software Maintenance And Reengineering*, 2003.

[8] 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.

[9] C. de la Higuera and F. Thollard. Identification in the limit with probability one of stochastic deterministic finite automata. In *Proc. of International Colloquium of Grammatical Inference and Applications*, 2000.

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

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

[12] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

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

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

[15] P. Hingston. Inference of regular languages using model simplicity. In *Proc. of the Australasian Conf. on Computer Science*, 2001.

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

[17] D. Lo and S.-C. Khoo. Quark: Towards better quality specification miners. *NUS tech. report, TRA 7/06*, 2006.

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

[19] L. Mariani and M. Pezze. Behavior capture and test: Automated analysis for component integration. In *Proc. of the Int. Conf. on Engineering of Complex Computer Systems*, 2005.

[20] 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.

[21] 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.

[22] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the Int. Conf. on Software Engineering*, 2001.

[23] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2003.

[24] Sun Microsystems,Inc. The java hotspot performance engine architecture. *online at http://java.sun.com/products/hotspot/whitepaper.html*, 1999.

[25] The Apache Software Foundation. Jakarta commons/net. *online at http://jakarta.apache.org/commons/net/*.

[26] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proc. of Int. Conf. on Software Engineering*, 2006.

[27] Y.Zou, T. Lau, K. Kontogiannis, T. Tong, and R. McKegney. Model-driven busineess process recovery. In *Proc. of Working Conf. on Reverse Engineering*, 2004.