SMArTIC: Specification Mining Architecture with Trace filtering and Clustering

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

ABSTRACT
Improper management of software evolution commonly leads to a lack of up-to-date specification. This situation is further aggravated by imprecise, changing requirements and short time to market requirement, which can result in software that is characterized by presence of bugs, anomalies and even security threat. Software specification mining is a new technique to address this concern by inferring specifications automatically. In this paper, we propose a novel specification mining architecture called SMArTIC (Specification Mining Architecture with Trace filtering and Clustering) to improve the accuracy, robustness and scalability of specification miners. This architecture is constructed based on two hypotheses: (1) Erroneous transactions should be pruned from traces input to a miner, and (2) Clustering related traces will localize inaccuracies in learning and reduce overgeneralization. Corresponding, SMArTIC comprises four components: an erroneous-trace filtering block, a related-trace clustering block, a learner, and a merger. We show through experiment that the quality of specification miner can be significantly improved using SMArTIC.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications; I.2.5 [Artificial Intelligence]: Programming Languages and Software—Software Protocol Specification Mining

General Terms
Dynamic Analysis, Automatic Reasoning, Machine Learning, Data Mining

Keywords
Clustering Traces, Filtering Errors, Specification Mining

1. INTRODUCTION

Improper management of software evolution commonly leads to a lack of up-to-date specification (cf. [7]). This situation is further aggravated by imprecise, changing requirements and short time to market requirement [4], which can result in software that is characterized by presence of bugs, anomalies and even security threats. There has been continual effort to develop techniques which aim to infer specifications automatically. In recent years, we have also seen a surge within the software engineering research community to adopt dynamic analysis, machine learning and statistical approach to address these problems, especially in the area of specification discovery [5, 10, 32, 1, 2, 9, 38, 17]. These methods are generally coined as specification miners. In [12], 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.

Specification miners can be classified into two groups, depending on how the mined specifications are represented: automaton-based [1, 5, 38, 32, 2] and non-automaton based [17, 9, 10, 31] specification miners.

The work by Ammons et al. has been considered the pioneer in automaton-based specification mining [1]. There, machine-learning approach is employed to discover program specification by analyzing program execution traces. 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.

In this paper, we leverage on the work performed by Ammons et al. in automaton-based specification mining, and explore the art and science behind the construction of such a miner. Specifically, we devise a novel architectural framework that achieves specification mining through pipelining of four functional components: Error-trace filtering, clustering, learning, and automaton merging. We demonstrate that such a system architecture improves the quality of mining result for two primary reasons:
1. Early identification and filtering of erroneous program execution traces can improve the quality of specification discovery.

2. Over-generalization occurred at learning stage can be mitigated by localization of learning process to groups of related program execution traces.

Contrary to other works done in automaton-based specification mining, we choose probabilistic FSA (PFSA) instead of (non-probabilistic) FSA as our learning target. PFSA is more expressive than FSA since it provides details on the probabilities of state transitions. This enables detection of frequently-used interaction patterns (e.g., "open (read)* close" pattern in a resource-access protocol) or sub-protocols within a specification, analogous to the idea of hotspots found in program execution [16].

The interactions are then recorded as program execution traces. We next invoke our miner to operate on this set of traces to produce an output specification. Lastly, we compare the output specification against the initial specification.

Our simulation model provides a controlled environment for experiment (cf., [23]). Given a piece of software component under experiment, we can alter the input API-interaction specification and later compare it with the corresponding output specification.

2.1 Specification Model

Input to our simulation is a specification model in the form of PFSA, a sample of which is shown in Figure 1. Based on this PFSA, we generates simulated traces. These simulated traces are then used to instruct the client to interact with the subject software, and the interactions are then recorded as execution traces. We refer the reader to Section 4 for a detailed description of this implementation.

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 error transitions (see the following paragraph), must have their probabilities summed up to 1.0.

The specification model can be injected with error by including error nodes and error transitions. Error transitions are modelled using dashed lines in Figure 1. 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 models (in terms of number of nodes or transitions) can be inputted to test the scalability of a miner. Hence, with injection of error and variety of model sizes different dimensions of quality assurance can be obtained.

2.2 Simulated-Trace Generation

In order to explore the effect of mining under different initial automata, we simulate the interaction with the software from these automata.
Actual program trace can be mapped to a string of alphabets as shown by Ammons et al. through 'standardization' process [1]. Strings of alphabets generated by our specification model (aka., simulated traces) can be considered as an abstraction of actual program execution traces; i.e., an alphabet representing a particular method call. Based on this abstraction, we generate simulated traces as strings of alphabets.

Simulated-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 emitting from the start node and sinking at the end node of a PFSA. We define an error trace as one that includes a transition sinking at an error node.

For each trace generated from a PFSA, we can determine its probability of being generated by multiplying together the probability of its constituents. We write $p(t)$ to denote the probability of a trace $t$.

To generate traces, we perform stratified random walk guided by the probability of PFSA’s transitions [25]. This 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.)

Traces will continue to be generated until all transitions have been covered at least $N$ times or $Max$ number of traces have been generated. By adjusting the value $N$, we can accommodate to slower learner that requires more than one trace from a specification to infer the automata model. By default we set $N$ to 10 and $Max$ to 10000.

Our algorithm is akin to the “code and branch coverage” criterion used in generating program test cases [13, 29]. Given a PFSA $M$ and a global percentage of error, our algorithm generates a multiset of traces $T$ possessing the following property:

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

This property ensures that all (non-error) transitions in $M$ have the opportunity to be used for trace generation. The algorithm detail is depicted in Figure 2.

$M_{EI}$ 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).

2.3 Precision, Recall and Co-emission

The terms precision and recall were originated from the field of information retrieval, where they are defined as “the proportion of retrieved documents which are relevant” and “the proportion of relevant documents retrieved” respectively [36].

Analogously, traces can be considered as documents and automata as pool/population of documents. Let original automata be denoted by $X$ and inferred automata by $Y$. Precision and recall can then be defined as the proportion of traces in $X$ that is accepted by $Y$ where $X$ is the original specification and $Y$ is the inferred specification.

The total number of traces accepted by an automata can possibly be infinite. Hence precision and recall can only be statistically approximated; here, Property 1 ensures that the set of traces generated are statistically sound.

In the context of PFSA, a trace might possibly be generated by both $X$ and $Y$, but their probability (of how frequently the trace will be generated) might differ greatly. Co-emission is therefore used address this probabilistic concern.

Co-emission has been used in measuring similarity between two Hidden Markov Models. Lyngso et al. propose several versions of similarity measurement [24]. One such metric which is adopted here, denoted by PS, provides an unbiased and normalized similarity measurement of two models $X$ and $Y$:

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

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

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

3. SMArTIC STRUCTURE
SMArTIC aims to increase a miner's precision, robustness and scalability by employing several novel techniques in specification mining. It leverages on the lessons learnt and experience accumulated from the past work done in this and related area (eg., [1], [10], [21], etc.). The success of SMArTIC hinges on the affirmation of the following two hypotheses:

**Hypothesis 1.** Mined specification will be more accurate when erroneous behavior is removed before learning than when they are removed after learning.

**Hypothesis 2.** Mined specification will be more accurate when it is obtained by merging the specifications learned from clusters of related traces than when it is obtained from learning the entire traces.

Hypothesis 1 is made from observing the system built by Ammons et al. [1]. In their work, a coring method is employed to remove erroneous transitions from the mined automaton. As this is performed on output automaton, erroneous transitions are included during mining. Consequently, performance of learning may be degraded. Moreover, pruning of transitions in an automaton may cause damage to the automaton, such as breaking an automaton into parts. This may then require substantial repairing of the automaton, and negates the effect of learning.

We believe that pruning of erroneous transitions should be done before learning. Consequently, we include a filtering process before learning process in SMArTIC, as we shall describe in Section 3.1.

Hypothesis 2 is made from the observation that existence of unrelated traces may negate the effect of learning via generalization; i.e., they can lead to over-generalization. Therefore, by clustering related traces and performing learning on each cluster, the effect of inaccuracies in learning can be localized to within a cluster. We believe this will result in a more accurate mined specification. Consequently, we include a clustering process in SMArTIC, as we shall describe in Section 3.2.

The overall structure of SMArTIC is as shown in Figure 3. It comprises 4 major blocks, namely filtering, clustering, learning and merging blocks. Each block is in turn composed of several major elements. Filtering block filters erroneous traces to address robustness issue. Clustering block divides traces into groups of 'similar' traces to address scalability issue. Learning block generates specifications in the form of automata. Merging block merges the automatons generated from each cluster into a unified one.

### 3.1 Filtering Block

Filtering block aims to filter out erroneous traces based on common behavior found in a multi-set of program traces. To filter well, we need a representation of common behavior which is intuitive enough to be used for filtering. Since a trace is basically a temporal or sequential ordering of events, representing common behavior by “statistically significant” temporal rules will be appropriate. Certainly, temporal rules based on full set of temporal logics will be a good candidate, but it is desirable to have a more light-weight solution.

Given a set of traces, we would like to generate, through mining, temporal rules of the following format: \( G(\text{pre} \rightarrow \text{post}) \), where \( \text{pre} \rightarrow \text{post} \) is of the following format

\[
\text{pre}_1 	imes \text{XF}(\text{pre}_2 \ldots \text{XF}(\text{pre}_{\text{end}} \rightarrow (\text{post}_1 \land \text{XF}(\text{post}_2 \ldots)))
\]

in Linear Temporal Logic (LTL) notation [15]. An example is \( G(a \rightarrow \text{XF}(b \land \text{XF}c)) \). The above basically says at any trace point when \( a \) occurs, \( b \) must eventually occur after \( a \), and \( c \) must also eventually occur after \( b \).

There are two commonly used measures of “statistical significance” namely, support and confidence (c.f [14]). Support of a rule \( G(\text{pre} \rightarrow \text{post}) \) is the number of trace points exhibiting the property \( \text{pre} \rightarrow \text{post} \). Confidence of the rule is the ratio of the number of trace points exhibiting the property \( \text{pre} \rightarrow \text{post} \) to those exhibiting the property \( \text{pre} \).

Rules having high confidence and reasonable support can be considered as “statistical” invariants. They thus characterize some general behaviors of a subgroup of traces. To detect outliers, only rules with high but less than 100% confidence will be useful. We call rules of \( G(\text{pre} \rightarrow \text{post}) \) format and exhibiting the above properties outlier detection rules.

Mined outlier detection rules will be used to filter out likely errors or unlikely behaviors. Any trace \( t_x \) of the following format \( a_1 \ldots a_i \ldots a_{\text{end}} \) will be filtered out (as an outlier) by a rule-set \( RS \) iff the following holds:

\[
\exists G(\text{pre} \rightarrow \text{post}) \in RS.
(\exists a_i.(\exists a_j. (1 \leq i \leq j) \land (a_1 \ldots a_j \text{ satisfies pre}) \\
\land (a_{j+1} \ldots a_{\text{end}} \text{ satisfies post})))
\]

The algorithm to filter out those traces that deviate from the general behaviors is depicted in Figure 3.1.

Implementation-wise, the structure of the filtering block is as shown in Figure 5. Outlier detection rules can be extracted efficiently by adding pre and post processing steps to a closed sequential pattern miner, BIDE [37]. The end result of the filtering block is a multi-set of filtered traces.

### Implementation Details.

Sequential pattern mining takes as input \( SA \) (a set of sequences) and \( \text{min_sup} \) (minimum support level). It will then reports subsequences (or pattern) contained by at least \( \text{min_sup} \) no of sequences in \( SA \). The number of such supporting sequences in \( SA \) is called ‘support’. Subsequences having support more than \( \text{min_sup} \) is called ‘frequent’. Sequential pattern miner will return both ‘frequent’ subsequences and its ‘support’.

Collected program traces, each of which is a sequence of method calls, can be considered as set of sequences \( SA \). Inputting collected program traces to a sequential pattern miner will generate a set of subsequences (or patterns) of method calls that is supported by many traces.

A program often contain loops. Subsequence (or pattern)
rather than no of supporting traces. Calls should be based on no of supporting

tern of method calls appear. 'Frequent' pattern of method calls is added to multiset

no of locations within traces ():

Rather than counting the no of traces, we should count the

of frequent subsequences (c.f. [39]). Long frequent subsequences of method call might appear due to ‘wrapper’ effect of deep class hierarchy, be an effect of decomposition of a complex methods to a series of simpler ones, or correspond to initialization and termination of a protocol (e.g. connect -> login -> logout -> disconnect in FTP protocol).

To reduce the combinatorial number of frequent subsequences, closed sequential pattern miner have been proposed [39]. A closed sequential pattern, is a frequent pattern which is not a subsequence of another frequent pattern with the same support. A set of closed sequential pattern captures the same information as full set of sequential pattern without being combinatorial in number. BIDE [37] is an optimised miner of closed sequential pattern. Experimental results have shown its run-time is linear w.r.t. input size (no of sequences x avg length of sequences). We run BIDE with preprocessed traces and support level Sup to generate a set of closed sequential patterns.

Frequent sequential patterns return by BIDE can be post-processed into pre→post rules. Given two frequent sequential pattern <A> and <A,B,C> with support s1 and s2, a rule A → BC can be generated with ‘confidence’ s2/s1 (c.f [34, 14]). Given a subsequence f which is frequent, support (f) = MaxcєClosed[f subsequence=of c] support (c). Hence, support of a frequent subsequence can be inferred from Closed. The conversion algorithm to generate rules from closed sequential pattern is as shown in Figure 6.

The algorithm receives as input a set of closed sequential patterns (Closed) and a minimum confidence level (Conf). It first build a prefix tree (or trie) of the closed sequential patterns. A prefix tree is a tree where there is one node for every common prefix. A closed sequential pattern can be considered a sequence of events. For each node q є trie, q.event, and q.prefix denotes its corresponding event and sequence of events from trie’s root to q. A node’s owner, q.owner, is the closed pattern sharing prefix q.prefix that has the maximum count. The support of q.prefix (denoted as q.count) is simply the count of q.owner. A closed pattern c ‘shared’ a trie node q, if q.prefix is a prefix of c.

The trie is later traversed to locate ‘interesting’ nodes. A node q is interesting if, ∃ qd. ((qd child-of q) ∨ (Conf ≤ (qd count/q.count) < 1)). From such nodes, rules with confidence ≥ Conf but < 1 of the form q.prefix → post, where post starts with qd.event can be generated.

The algorithm will extract pre→post rules having confidence at least Conf but less that 100%. All rules generated will also be frequent since it is generated from closed sequential patterns. Some rules that can be inferred from others were pruned since they are redundant. The details of pruning and completeness issue are outside the scope of this technical report.

By trace pre-processing, rules mined will apply at every trace points. Hence, each rule can be expressed as G(pre→post), where pre → post is of the following format pre1∧XF(pre2 …∧ XF(preend → (post1 ∧ XF(post2…)))) in LTL notation. These rules will have support of at least Sup and confidence ≥ Conf while < 100% – i.e. they are outlier detection rules. Hence, we have mined outlier detection rules efficiently by adding pre and post processing steps to a closed sequential pattern miner.

---

**Figure 4: Filtering Traces from Temporal Rules**

**Figure 5: Filtering Block Structure**
3.2 Clustering Block

Input traces might be “mixed up” from several rather unrelated scenarios, e.g., group of related traces that represent a usage pattern of an API/component. Grouping unrelated traces together for a learner to learn might multiply the effect of inaccuracies in learning a scenario. Such inaccuracies can be further permeated into other scenarios through generalization.

Clustering block converts a set of traces into groups of related traces. Clustering is meant to localize inaccuracies in learning one sub-specification and prevent the inaccuracies from being permeated to other sub-specifications. Furthermore, by grouping related traces together, better generalization (aka., less over-generalization) can be achieved when learning from each cluster.

Two major issues pertaining to clustering are: the choice of clustering algorithm and an appropriate similarity metrics; i.e., measurement of similarity between two traces. Performance of clustering algorithm is affected by appropriate similarity/distance metrics used. Different clustering algorithms learn differently in terms of accuracy, efficiency and the level of user interaction required. (*c.f. [14]) The general structure of clustering block is as shown in Figure 7.

3.2.1 Clustering Algorithm

We use a classical off-the-shelf clustering algorithm for our purpose, namely k-medoid algorithm [19]. K-medoid

\[\text{Procedure Generate Rules} \]

**Inputs:**
- **Closed:** A set of closed sequential patterns (seq. of events)
- **Conf:** Minimum confidence

**Outputs:**
- **Rules:** Outlier detection rules

**Method:**

**Step 1: Trie building and traversal**  
Let Trie = Build a prefix tree from Closed

For each node p in Trie do
  - Set p.prefix = Seq of events from root to p (inclusive) in Trie
  - Set p.owner = f, where f ∈ Closed ∧ f shared p
  - Set p.postfix = Seq of events from p to leaf, where seq from root—p—leaf generates p.owner
  - Set p.count = p.owner.count

Let Interesting = \{q| q ∈ Trie ∧ \exists qd.(qd child-of q) ∧ (Conf ≤ q.count/qd.count < 1)}

**Step 2: Rule generation**

For each q ∈ Interesting do
  - Let qdesc = \{qd | qd child-of q} ∧ (Conf ≤ q.count/qd.count < 1)
  - Let RS = Generate rules pre → post where,
    - pre = q. prefix ∧ post qdesc.postfix
  - Add RS to Rules

**Output Rules**

![Figure 6: Rule Generation](image)

![Figure 7: Clustering Block Structure](image)

algorithm works by computing the distance between pairs of data items based on a similarity metric; this corresponds to computing distance between pairs of traces. It then groups the traces with small distances apart into the same cluster. k in k-medoid is the number of clusters to be created.

In our implementation, we adapt the Turn* algorithm presented by Foss et al. [11] into the k-medoid algorithm. The Turn* algorithm can automatically determine the number of clusters to be created by considering the similarities within each cluster and differences among clusters. Our algorithm will repetitively increase the number of clusters. For each repetition, it will divide datasets into clusters and evaluate a measure of similarities within each cluster and differences among different clusters. The algorithm will terminate once local maxima is reached. The algorithm is shown in Figure 8. At program point (*) a call is made to procedure CalculateSim defined in subsection 3.2.2.

3.2.2 Similarity Metric

In many applications, comparison between two data items is relatively clear – sometimes it only involves a simple subtraction of two numbers – e.g., (Avg. profit for company x) - (Avg. profit for company y), etc. However, comparison of two program traces is neither so clear cut nor easily obtained.

Our first idea is to use global sequence alignment [35] to measure the distance between two traces. This alignment is frequently used to obtain similarity metrics of two DNA sequences. Its main idea is to insert ‘dash’ or spaces within strings to obtain the most accurate matching of two strings. Different from Knuth-Morris-Prat (KMP) algorithm [6], the sequence alignment algorithm finds the best approximated alignment(s) rather than the occurrence of exact match. Alongside the best alignment(s), an overall similarity score will also be reported. We use this score as the similarity metric between the two program traces.

This first idea does not work well in practice because, contrary to normal strings, program traces exhibit some characteristic which makes it difficult to measure similarity by a simple alignment of two traces. Specifically, a trace might only be different to another due to different numbers of loop iterations during program execution. As an example, con-
3.3 Learning Block

Although temporal rules have also been used ([40]) to capture certain information of a program specification, automata have been commonly used in capturing specifications, especially protocol specifications. The purpose of this learning block is to learn automata from clusters of filtered traces. This block is actually a placeholder in our architecture. Different PFSA specification miners can be placed into this block, as long as they meet the input-output specification of a learner. Once a learner is plugged in, it will be used to mine the traces obtained from each cluster. At the end, the learner produces one mined automaton for each cluster.

In the current experiment, we choose to use a PFSA specification miner that has been used for software specification mining earlier, i.e. sk-strings learner [30].

Sk-strings learner is used by Ammons et al. to mine the specification of X11 windowing library [1]. It is an extension of the k-tails heuristic algorithm of Biermann and Feldman [3] for learning stochastic automata. In k-tails, two nodes in a constructed automata are checked for equivalence by looking at up to subsequent k-length strings that can be generated from them. Different from k-tails, in sk-strings, subsequent strings need not necessarily end at an end node, except for strings of length less than k. Furthermore, only the top s% of the most probable strings that can be generated from both nodes are considered. Implementation-wise, the sk-strings learner first builds a canonical “machine” similar to a prefix-tree acceptor from the traces. The nodes in this canonical machine are later merged if they are indistinguishable with respect to the top s% of the most probable strings of length at most k that can be generated starting

Suppose that APICLIENT_ABCD is a client function of an API. It is conceivable that the API interaction patterns for various runs via function call APICLIENT_ABCD with different input parameters should be grouped together. So, for a run with parameters outer_iter = 2 and inner_iter = [2,3], the generated trace is ABCBCDABCBCBCD. For another run with outer_iter = 1 and inner iter = [1], the generated trace is ABCD. Now, if we simply align these two strings, even in their best alignment their similarity score will be too low for them to be grouped into the same cluster.

Our solution to the above problem is to instead compare the regular expression representations of the two traces rather than their actual sequence of alphabets. Converting to its regular expression, the first trace will be (A(BC)+D)* which corresponds closely to ABCD. Note, we choose to only use “*” quantifier (i.e. one or more repetitions) but not “+” quantifier since every expression can be reduced to ((a1)* (a2)* ... (an)*)*, where n ≥ 0, a1, a2, ..., an ∈ Alphabet.

We obtain the regular expression representation by converting a trace to its hierarchical grammar representation using Sequitur [27]. Output of sequitur will be used for construction of regular expression and then be fed in as input to global sequence alignment. With these we obtain a method to find a reasonable distance metrics on similarity of program traces. Generated sequitur grammar of a trace will be processed through several passes of reduction to produce a regular expression representation. The distance calculation algorithm and grammar reduction to regular expression algorithm are shown in Figure 9. It makes use of HieUni algorithm that is shown in Figure 10.

Figure 8: PLess KMedoid Algorithm

The following program segment:

```
    function APICLIENT_ABCD (outer_iter, inner_iter[]) {
        for (int j=0;j<outer_iter;j++) {
            for (int k=0;k<inner_iter[k];k++) {
                do( k++;
                Call API.A();
                Call API.B();
                Call API.C();
                Call API.D();
            }
        }
    }
```

```
3.4 Merging Block

The merging process aims to merge multiple PFSA’s produced by the learner into one such that there is no loss in precision, recall and likelihood before and after the merge. Equivalently, the merged PFSA accepts exactly the same set of sentences with the same probabilities as the combined set of sentences accepted by the multiple PFSA’s.

The primary purpose of the merging process is to reduce the number of states residing in the output PFSA by collapsing those transitions behaving “equivalently” in two or more input PFSA’s, thus improving scalability.\(^2\)

A secondary objective is to enable comparison of structural similarity between the mined PFSA and the original PFSA.

We omit the detail in this paper.

\(^2\)A secondary objective is to enable comparison of structural similarity between the mined PFSA and the original PFSA.

\[ t_{12} \cdots t_{m} \overrightarrow{\delta} t_{m+1} \cdots t_{p}, \]

A suffix of \( n_{1} \) is the string \( \delta t_{m+1} \cdots t_{p} \), and a prefix of \( n_{1} \) is the string \( t_{12} \cdots t_{m-1} \).

In the case where no transition emitting from \( n \) appears in the string, both the prefix and suffix of \( n \) with respect to that string are just null.

Extending from the definitions above, the set of prefix/suffix of \( n \) in a PFSA is the set of prefix/suffix of \( n \) with respect to all strings accepted by the PFSA.

The definition of equivalent transitions above admits transitive behavior: If two transitions are equivalent as defined by sharing the same set of suffix, then each of the transitions in the suffix-set is also an equivalent transition. The similarity between the mined PFSA and the original PFSA. We omit the detail in this paper.
same behavior can be observed from equivalent transitions sharing the same set of prefix.

The Merging process also ensures that the likelihood of traces generated by the output PFSA remains the same as that of the combined input PFSA's. More specifically, let \( \mathcal{A} \) be the output PFSA and \( \mathcal{A}_i \) (\( i = 1 \ldots n \)) be the input PFSA's. Let \( \delta \) be a transition in \( \mathcal{A} \), then
\[
p_{\mathcal{A}}(\delta) = \sum_{i=1}^{n} w_i \cdot p_{\mathcal{A}_i}(\delta)
\]
where \( p_{\mathcal{A}}(\ell) \) and \( p_{\mathcal{A}_i}(\ell) \) represent the probabilities of the transition \( \ell \) located in PFSA \( \mathcal{A} \) and its equivalent transitions occurring in \( \mathcal{A}_i \). \( w_i \) is the weightage given to each cluster hosting their own input PFSA; it is the ratio of the number of traces in the cluster to the number of total traces in the entire system.

Implementation-wise, the transitivity property of equivalent transitions enables an incremental detection of such transitions, starting from either the start node (for finding prefix) or the end node (for finding suffix) of a PFSA.

**Implementation Details** Merger block will merge PFSA's produced by learner block to a unified one. The merging process is performed in the “safe” way where no further generalization is performed during the merging process. Hence the set of language accepted by the final PFSA will be the union of the set of languages accepted by each of the PFSA's. The merging process is performed iteratively by merging 2 PFSA's at a time. Let’s call the them X and Y for ease of reference. The algorithm for automaton merger is as shown in Figure 11.

There are 6 steps in automaton merge algorithm. They are (1). Handling of exception cases, (2). Creation of unifiable list, (3). Creation of mergable list, (4). Creation of equivalence class, (5). Creation of structural merge and (6). Creation of full merge.

We consider it an exception case if there is a transition sinking in start node. This is such since the algorithm assume that the start nodes of the two automatons are unifiable/mergable. However, if one automaton has start node as sink node of a transition and not the other, the two start nodes are not unifiable. In step 1, if this exception case happens a new start node is added with \( \epsilon \)-transition to the original start node.

Two nodes of X and Y are considered unifiable if they shared common set of prefixes. In step 2, we create a list of pairing between \( x \in X . N o d e s \) to \( y \in Y . N o d e s \) where \( x \) and \( y \) are unifiable (i.e. creation of unifiable list). Two nodes of X and Y are considered mergable if they shared common set of suffixes. In step 3, we create this list of pairings (i.e. creation of mergable list).

The two lists are created by generating and solving constraints in a similar way with only minor differences. Unifiable list is created by unifying starting from start node top-down while mergable list is created by unifying starting from end node bottom-up. To check whether xNode and yNode are unifiable, only transitions having xNode or yNode as their source (i.e. x.Next and y.Next) need to be considered. Vice versa, only transitions having xNode or yNode as sink (i.e. x.Prev and y.Prev) need to be considered for xNode and yNode to be mergable.

A node can be unifiable to more than one node. In step 4, we would like to create these equivalence classes. A representative node is created for each equivalence class. Since two mergable nodes shared a common set of suffixes, it must be the case that all their descendants (i.e. nodes reachable from them) must be mergable as well. Hence we can create transitions between representative nodes. The end result is a sub-automaton at the bottom of the merged automaton.

In step 5, we create a partially merged automaton by traversing from root node according to unifiable list created in step 2. In this step, we create the sub-automaton at the top of the merged automaton.

In step 6, we augment this partially merged automaton by adding in nodes and transitions in X and Y that has not been included to the merged automaton. If a transition leads to a mergable node m, we add a transition to the m’s representative node in sub-automaton created in step 4 – no descendants of m need to be traversed anymore.

---

**Procedure AutomatonMerge**

**Inputs:**
- \( X \): First automaton to merge
- \( Y \): Second automaton to merge
- \( \text{TrainSetX} \): Set of traces to train X
- \( \text{TrainSetY} \): Set of traces to train Y

**Outputs:**
- \( \text{Merged} \): Merged automaton with \( L(\text{Merged}) = L(X) \cup L(Y) \)

**Method:**

1. **Step 1: Handle Exception Cases**
   - If \( \#X.\text{StartNode}.\text{Prev} = 0 \) 
   - X.AddTempStartNode()
   - If \( \#X.\text{StartNode}.\text{Prev} = 0 \) 
   - X.AddTempStartNode()

2. **Step 2: Create Unifiable List**
   - Let \( \text{Uni} = \text{Call CreateUnifiableList(X,Y,T)} \)

3. **Step 3: Create Mergable List**
   - Let \( \text{Merge} = \text{Call CreateUnifiableList(X,Y,F)} \)

4. **Step 4: Create Equivalence Class**
   - Let \( \text{DSSet} = \) 
     - Call CreateEqClass (Merged,RepObjSet)

5. **Step 5: Create Structural Merge**
   - Let \( \text{root} = \) 
     - Create new node 
     - Call CreateStructuralMerge 

6. **Step 6: Create Full Merge**
   - Call CreateFullMerge 

**Figure 11: Automaton Merger**

After the above 6 steps, the result is a merged automaton which is “safe” where no additional generalization was made during the merging process. The algorithm refers to unspecified procedures. These procedures detail can be found in the Appendix.
4. JAKARTA COMMONS NET

Jakarta Commons Net [26] is a set of reusable open source java code implementing the client side of many commonly used network protocols. We built a simple CVS (Concurrent Versions System) functionality on top of FTP library provided by Jakarta Commons Net.

4.1 Protocol for CVS-FTP API Interaction

This CVS functionality can be considered a client of Jakarta Commons Net having a certain protocol pattern. Our CVS class and Commons Net library can be instrumented to generate traces which were then inputted to SMARTrIC and sk-strings. The resultant model are then compared with the original CVS specification to evaluate the feasibility of SMARTrIC in improving the accuracy of the results over the sk-strings learner.

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 12. The dashed boxes, from top to bottom, represent upload files, initialization, delete files, make directories, remove directories and download files scenario respectively.

4.2 Instrumenting Jakarta Commons Net

We instrument Jakarta Commons Net with JRat, the Java runtime analysis toolkit [18]. The instrumentation is composed of an instrumentation byte code injection and a trace collection part. By default, JRat logs execution traces by associating them with a localized context. This context is simply a list of method calls in the runtime stack (ie., main () -> FTPClient.<init> -> TelnetClient.<init>). Information having the same context is grouped together. Given a class file to instrument, JRat will add instrumentation code to all methods except the constructor.

We modified JRat core classes and added a plug-in to it. JRat (with default plug-ins) satisfies most of our tracing requirements except for the following:

Capturing order of method calls along with context. We would like to capture the temporal order of method calls together with the context. However, JRat may destroy the order of method calls in the context. Information on calls to a method at two different times under the same context but with different temporal ordering should not be grouped together.

Instrumentation of class constructor. In order to capture the hierarchy of method calls well, we need to instrument the class constructor as well. Class constructor might calls other methods. We would like to capture the information that their context are the same but different from method calls called at constructor context.

Capturing object identities. Our plug-in captures object identities through their hash-code as well as the original class names where they are instantiated from (ie., rather than the parent class where a called function is defined). The JRat structure provides good support for these information.

Thread slicing and Scalability. We slice traces into threads and generate separate trace file for each of them. For scalability, no large trace related information are stored in memory. They are always output incrementally.

The result of JRat instrumentation is an injection of tracing byte code to java classes. Running the instrumented code will produce a tree of method calls capturing their order and context represented as an XML document.

4.3 Trace Collection and processing

We construct a wrapper class that takes in the automata shown in Figure 12 and invokes org.apache.commons.net.ftp. FTPClient accordingly. The wrapper class will generate a list of sequences of method invocations by traversing the automata from the start to the end node multiple times until coverage criteria is met. The result is a simulation of a regression testing of CVS-FTP API interaction.

Each invocation of a method of FTPClient may generate exceptions, especially FTPConnectionClosedException and IOException. Hence the code accessing the FTPClient methods need to be enclosed in a try..catch..finally block. Every time such exception happens we simply logout and disconnect from the FTP server. This is simulated by adding error transitions as shown in Figure 13. Ten percent error is assumed and erroneous trace will be injected to 10% of the generated list of sequences of method invocations.

The trace file generated is likely to be huge because of the wrapper effect and long class hierarchies. On the other hand, what we really need are the traces capturing interaction between our own CVS classes and FTPClient. To get that, we process the trace file as follows : (1) Traverse the XML trace file depth-first, and locate all the first invocations of the client method calls. Each such location will correspond to a scenario trace sequence. (2) From the above locations, traverse depth-first, and locate all the first invocations of the API method calls. The API method calls might not be directly below the client method calls in the trace file XML hierarchy.

4.4 Protocol Specification Generation and Results

The collected traces are inputted to different miners: SMARTrIC (with sk-strings in the learner block), SMARTrIC with-
out filtering, SMARToTIC without clustering, sk-strings with coring and standalone sk-strings. Coring threshold is set at 0.2 level. SMARToTIC filtering confidence and support is set at 0.8 and 0.1 respectively. Default parameter settings is used for sk-strings both when standalone and within SMARToTIC.

Protocol specification is then produced and compared against the original one (as shown in Figure 12) in terms of precision and recall. We repeat the above experiments 100 times using different lists of scenario trace sequences.

The following table shows the results of our experiment. The columns Precs, Recall and PS correspond to precision, recall and unbiased, normalized co-emission respectively (as defined in subsection 2.3).

<table>
<thead>
<tr>
<th></th>
<th>Precs</th>
<th>Recall</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMARToTIC</td>
<td>0.484</td>
<td>0.981</td>
<td>0.653</td>
</tr>
<tr>
<td>SMARToTIC w/o filtering</td>
<td>0.426</td>
<td>1.000</td>
<td>0.616</td>
</tr>
<tr>
<td>SMARToTIC w/o clustering</td>
<td>0.263</td>
<td>0.984</td>
<td>0.532</td>
</tr>
<tr>
<td>sk-strings (coring)</td>
<td>0.289</td>
<td>0.581</td>
<td>0.447</td>
</tr>
<tr>
<td>sk-strings</td>
<td>0.225</td>
<td>1.000</td>
<td>0.583</td>
</tr>
</tbody>
</table>

As shown in the table, SMARToTIC improves the precision and co-emission while maintaining good recall of CVS protocol inference. The precision of SMARToTIC are more than double the precision of sk-strings.

Both filtering and clustering help in increasing precision while maintaining good recall or recall and unbiased, normalized co-emission respectively.

The drawback of coring is shown in the results where recall drops by almost half. Although precision is increased, there is a heavy penalty in recall: Pruning erroneous behaviors indvertibly removes a significant proportion of correct behaviors.

It is also of interest to know the number of erroneous traces our filtering algorithm filters out. On the average it filters out 43% of erroneous traces while only 4% of valid ones.

We have conducted thorough experiments using this application to verify both our hypotheses. Evaluating the statistical significance of our experiment results, we perform one-tailed t-test on the experiment results. The result is shown in table below.

<table>
<thead>
<tr>
<th>H0</th>
<th>Result</th>
<th>Conf L</th>
</tr>
</thead>
<tbody>
<tr>
<td>mr(SMARToTIC w/o clus) = mr(cor)</td>
<td>R</td>
<td>99.99%</td>
</tr>
<tr>
<td>mp(SMARToTIC w/o clus) = mp(cor)</td>
<td>NSE</td>
<td>90%</td>
</tr>
<tr>
<td>mps(SMARToTIC w/o clus) = mps(cor)</td>
<td>R</td>
<td>99.99%</td>
</tr>
<tr>
<td>mp(SMARToTIC w/o fil) = mp(str)</td>
<td>R</td>
<td>99.99%</td>
</tr>
<tr>
<td>mr(SMARToTIC w/o fil) = mr(str)</td>
<td>NSE</td>
<td>90%</td>
</tr>
<tr>
<td>mps(SMARToTIC w/o fil) = mps(str)</td>
<td>R</td>
<td>90%</td>
</tr>
</tbody>
</table>

mr, mp and mps denotes mean recall, mean precision and mean unbiased, normalized co-emission respectively. SMARToTIC w/o clus, SMARToTIC w/o fil, cor and str denotes SMARToTIC without clustering (only filtering on), SMARToTIC without filtering (only clustering on), sk-string with coring and sk-string (standalone) respectively. R and NSE denotes rejected and not sufficient evidence respectively.

The first result shows that it is statistically significant that mean recall of SMARToTIC w/o clustering (only filtering on) is higher than mean recall of coring. The second result shows that mean precision of SMARToTIC w/o clustering is not statistically different from mean precision of coring. The third result shows that co-emission of SMARToTIC w/o clustering is higher than co-emission of coring. Hence, mined specification will be more accurate (improved recall and co-emission with equivalent precision) when erroneous behavior is removed before learning (in SMARToTIC w/o clustering) than when they are removed after learning (in sk-string (with coring)). This supports our first hypothesis (stated in Section 3).

The fourth result shows that it is statistically significant that mean precision of SMARToTIC w/o filtering (only clustering on) is higher than mean precision of sk-string (standalone). The fifth result shows that mean precision of SMARToTIC w/o filtering is not statistically different from mean precision of sk-string (standalone) – precision of both are exactly the same with mean 1 and standard deviation 0. The sixth result shows that co-emission of SMARToTIC w/o filtering is higher than co-emission of sk-string (standalone). Hence, mined specification will be more accurate (improved precision and co-emission with equivalent recall) when it is obtained by merging the specifications learned from clusters of related traces than when it is obtained from learning the entire traces. This supports our second hypothesis (stated in Section 3).

5. FURTHER EXPERIMENTS

The experiment with CVS specification in Section 4 provides a positive evidence that SMARToTIC is a feasible architecture for improving mining accuracy; it also provides strong evidence to support our hypotheses stated in Section 3.

In this section, we perform further experiments on SMARToTIC, not just on its accuracy, but also on its robustness and scalability. To this end, we have conducted almost 2000 experiments to support the superiority of SMARToTIC.

Our experiments remains to be over the same set of miners, with sk-strings learner being employed either in standalone mode or in corporation with other processes, especially as the learner block of SMARToTIC.

5.1 Material

In the first set of experiments, two sets of sub-experiments using different types of error injection were performed to evaluate the two learners’ performance in terms of robustness. These experiments are performed on sk-strings (standalone), sk-strings (with coring) and SMARToTIC. For SMARToTIC case, we disable the clustering sub-system to measure the effect of filtering block.

We simulated the automaton generated by Ammons et al. in their analysis of X11 windowing library (cf. [1]) – as shown in Figure 14. However, we modified the model slightly so that it was without any non-determinism and re-
peated use of alphabet assigned to transitions and we added probabilities. Probabilities are distributed equally to transitions from the same source node (not shown in the diagram). This is meant to produce a base model that can be learned perfectly. Error nodes and transitions were then injected to the automaton to conduct robustness tests. The models with different injections of errors (nodes and transitions labelled as Z and shown with dashed lines) are shown in Figure 15(a) & (b). Each of the two types of injections of errors shown in Figures 15(a) and (b) respectively corresponds to a separate set of sub-experiments.

Figure 14: X11 Windowing Library Model

Figure 15: Models of Specification with Error

We expect the specification miners to be able to filter out errors. We compared the inferred automaton with the simulator model shown in Figures 15(a) and (b) without error nodes and transitions and recorded the similarity and difference metrics. We generated traces using trace generation algorithm (briefly mentioned in subsection 2.1) and capped the maximum number of traces generated to 10,000. Four, eight and ten percents of error were injected to the system (ie., 4, 8 and 10 percent of generated traces respectively will be erroneous). We assume the error level is unknown to the learner except that it's low. Hence, the threshold used for coring was set to 0.2. SMArtIC’s filter confidence is also set at an equivalent level of 0.8 while its support is set at 0.1. In each case, we ran a hundred experiments and recorded the average performance.

In the second experiment, we evaluated the scalability of the learners by generating distinct models of various sizes. Two sets of sub-experiments were conducted, each with a different independent variable. In the first set, we varied the number of nodes (by 10, 20, 30 and 40) in the specification model and maintained the number of outgoing transitions per node to at most four. In the second set, we varied the number of outgoing transitions per node (by 3, 5, 7 and 9) and maintained the number of nodes at 10. For each case, we performed 10 experiments and recorded their average performance.

We automatically generated distinct models having n nodes and maximum of m transitions per state with common start and end nodes. Transition labels were chosen from a pool of fixed number of labels randomly. Loops were 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 (eg., business logic of an enterprise application).

Our model generation algorithm is shown in Figure 16. 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.

![Figure 16: Model Generation Algorithm](image)

These experiments were performed on sk-strings (standalone) and SMArtIC. In the SMArtIC case, we measure the effect of clustering block by disabling filtering sub-system.
We generated traces and capped the maximum no of traces at 10,000. No error was injected to the system. Since we imposed a cap of 10,000 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.

5.2 Experiment 1 Findings
Here, we evaluated the robustness of sk-strings, sk-strings (coring) and SMARtic with two different injection of errors. The models with different injection of errors are shown in Figures 15(a) and (b).

Results. The experiment results for ErrModel1 and ErrModel2 are captured below. Column E% corresponds to the percentage of erroneous traces. Columns Prec, Recall and PS correspond to precision, recall and unbiased, normalized co-emission respectively (as defined in subsection 2.3).

<table>
<thead>
<tr>
<th>Error Model 1</th>
<th>sk-strings</th>
<th>sk-strings(coring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E%</td>
<td>Prec</td>
<td>Recall</td>
</tr>
<tr>
<td>4</td>
<td>0.946</td>
<td>1.000</td>
</tr>
<tr>
<td>8</td>
<td>0.988</td>
<td>1.000</td>
</tr>
<tr>
<td>10</td>
<td>0.883</td>
<td>1.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Model 2</th>
<th>sk-strings</th>
<th>sk-strings(coring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E%</td>
<td>Prec</td>
<td>Recall</td>
</tr>
<tr>
<td>4</td>
<td>0.999</td>
<td>1.000</td>
</tr>
<tr>
<td>8</td>
<td>0.993</td>
<td>1.000</td>
</tr>
</tbody>
</table>

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 slightly larger to the noise injected (5.4%,5.3%) vs. 4%, 9.2%,10.2% vs. 8% and (11.7%,12.5%) vs. 10%); learner precision degrades in the presence of erroneous traces. We conclude that the sk-strings learner is not robust.

SMARtic result is similar to sk-strings in that all traces generated by the simulator model X were 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 ((0.1%,0.6%) vs. 4%, (0.7%,1.4%) vs. 8% and (1.9%,2.6%) vs. 10%). These indicates that filtering of erroneous traces is effective in preventing loss of precision.

The most important observation here is that: Having coring as post-processing to sk-strings removes not just erroneous transitions but also quite a fair number of correct transitions. Consequently, the accuracy of the mined specification degraded. This result strongly supports our first hypothesis.

Another limitation of coring is due to the fact that transition labels are being ignored during the coring operation. Coring method only searches for the pair of nodes (i,j) where there is a low “heat transmission” from node i to node j [1]; it ignores the detail of how the node j is reached (which can be single transition, set of transitions, single path or set of paths). In the second sub-experiment, erroneous transitions go to valid nodes instead of special error node. This results in little/no filtering when coring is used.

5.3 Experiment 2 Findings
We performed two sets of scalability sub-experiments. In the first set of sub-experiments, we generated distinct models by varying the number of nodes while keeping max transitions per node at 4. In the second set of sub-experiments, we varied max number of transitions while keeping the total number of nodes constant at 10.

Results. The experiment results is shown below for sk-strings and SMARtic.

### Varying No of Nodes

<table>
<thead>
<tr>
<th>Varying No Of Transitions</th>
<th>sk-strings</th>
<th>SMArtic</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.N,X.TN</td>
<td>Prec</td>
<td>Recall</td>
</tr>
<tr>
<td>10/4</td>
<td>0.947</td>
<td>1.000</td>
</tr>
<tr>
<td>20/4</td>
<td>0.999</td>
<td>1.000</td>
</tr>
<tr>
<td>30/4</td>
<td>0.999</td>
<td>1.000</td>
</tr>
<tr>
<td>40/4</td>
<td>0.999</td>
<td>1.000</td>
</tr>
</tbody>
</table>

### Varying Max No Of Transitions

<table>
<thead>
<tr>
<th>Varying Max No Of Transitions</th>
<th>sk-strings</th>
<th>SMArtic</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.N,X.TN</td>
<td>Prec</td>
<td>Recall</td>
</tr>
<tr>
<td>10/4</td>
<td>0.113</td>
<td>1.000</td>
</tr>
<tr>
<td>10/5</td>
<td>0.187</td>
<td>1.000</td>
</tr>
<tr>
<td>10/10</td>
<td>0.844</td>
<td>1.000</td>
</tr>
<tr>
<td>10/9</td>
<td>0.973</td>
<td>0.997</td>
</tr>
</tbody>
</table>

Analysis. The above results shows that sk-strings and SMARtic were affected when we scaled up the model size. Comparing the two set of experiments, we observe that the precision is adversely affected in all cases when we increase the number of nodes, whereas the impact is less severe when we increase the number of transitions.

SMARtic is generally better in terms of precision up to a factor of over 147.5 (i.e., 30-node case). In the second set of experiments (i.e., when we increase the maximum number of transitions), SMARtic maintains its precision while sk-strings loses it as max no of transitions increased.

6. RELATED WORKS

Specification Miners. 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, 5, 38, 32, 2] and non-automaton based [40, 9, 10, 31] specification mining.

The specification miner described in [1] has been extensively referred in this paper. In other work, Whaley et al. extract object-oriented component interface sequencing constraints to form multiple finite state automaton [38]. Reiss et al. encode program execution traces as directed acyclic graph to aid visualization and understanding of program [32]. Arts et al. dynamically extract program models from Erlang program as state graph model for model checking and visualization [2]. We believe that these and other similar miners can benefit from our architecture with minimal changes.

Mining Rules for Software Engineering Tasks. In [9], association rules (i.e., describing association of items rather than sequence of events) based on frequent itemset mining
have been used to describe user interaction behavior for web-based GUI.

In [40], Yang et al. present their work in mining response pattern [8] using a set of templates. They only consider rules involving two events; i.e., of the form $a \rightarrow b$. In order to handle longer rules, Yang et al. introduce “chaining”. For example, if both $A \rightarrow B$ and $B \rightarrow C$ are significant, they can be concatenated to form $A \rightarrow B \rightarrow C$ which will be significant too. However, the reverse may not be always true: $A \rightarrow B \rightarrow C$ might be significant although only rule $A \rightarrow B$ is significant while $B \rightarrow C$ is not. For such cases, the rule $A \rightarrow B \rightarrow C$ cannot be generated by inferring from two-event rules and chaining them. Hence, chaining only gives partial solution rule set for multi-event (>2) sequential patterns.

SMArTIC also mines temporal response pattern [8] but for the purpose of generating rules for outlier detection. In our case, expressivity is required. Thus, the pre- and post-condition of our mined outliers detection rules can take in any number of events. This pattern is also referred to as chain response pattern in [8]. Moreover, rather than mining all possible statistically significant rules, which might be of combinatorial number (see discussion on closed sequential patterns in [39]), we only mine those useful for outlier detection.

To further enhance the rule expressivity, Yang et al. add additional constraints to response pattern, the strictest of which is an “alternating” pattern described in regular expression as $(ab)^*$ or in LTL as $G(a \rightarrow X \bar{F} b) \land (\neg bUa) \land G(a \rightarrow X(\neg aUb)) \land G(b \rightarrow X(\neg bUa))$ – the LTL expression after the first $\land$ is the additional constraint. Hence, both approaches have their own merits.

**Program Trace Characterization.** Larus proposes using whole program path (WPP) to represent a program’s dynamic control flow [22]. Sequitur [27] is used to compact acyclic program trace into WPP which is basically its corresponding grammar. WPP is later used to find hot subpaths – heavily executed code – for optimization purpose.

Reiss et al. present a system for analyzing java trace data [33]. Sequitur [27] is used during path analysis to compact a trace into its grammar representation which serves as its identifier.

Similar to Larus and Reiss et al., we used Sequitur to get a representation of program trace. In many applications, Sequitur is able to infer reasonable hierarchical structure from a sequence of discrete symbols. Different from Larus and Reiss et al., our purpose is to obtain an intuitive measure of similarity between two traces while taking their inherent structure into consideration. Hence, rather than taking grammar output of sequitur directly, we post-process it into a regular expression (regex) format. The regex representation ‘flatten’ the grammar while retaining its structure. Two regex representation can then be easily compared by global sequence alignment which produces their best alignment.

**Clustering Temporal Data & Similarity Measures.** In clustering temporal data, there are numerous works on clustering time series data and similarity measures between two time series (c.f. [20]). However, there are huge difference between time series data – e.g. stock market data – and program trace data. The first are often lacking in structure while the later have underlying structure since they are generated from program having structured control flow.

### 7. CONCLUSION

In this paper, we began with two hypotheses about how specification miners should be organized to alleviate the impact of erroneous transitions and to localize and minimize over-generalization. We then presented our novel Specification Mining Architecture with Trace filtering and Clustering, (SMArTIC) to support our hypotheses. SMArTIC comprises four major blocks – clustering, filtering, learning and merging. Filtering and clustering is meant to address the issue of robustness and scalability respectively.

Traces deviating from common trace population rules are removed. The resultant filtered traces are then separated into multiple 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 filtered traces are then inputted to a specification miner. The sk-strings learner is used for learning, and each cluster is considered an independent (sub-)protocols. Lastly, a merger sub-system produces a merged automaton without sacrificing accuracy.

Along with the architecture, we have also proposed a novel trace clustering technique based on grammatical similarity, a novel outlier detection rules mining technique and a novel automaton merging method. Besides having automaton as specification, the outlier-detection rules produced by the filtering block can also be viewed as sets of simple specifications based on strong properties of significant trace group useful for filtering. They can effectively capture those property pattern proposed in [8] which are interesting for program traces and useful for identifying potential bugs.

We experimented with Jakarta Commons Net open-source library. Our experiments aim at deriving API interaction protocol for client application of Jakarta Commons Net open-source library [26]. From one hundred experiments performed, the following are noted: (1) SMArTIC improves precision (more than double) and co-emission while maintaining good recall (2) Both clustering and filtering help in improving precision while maintaining good recall and equivalent co-emission (3) Coring removes erroneous behavior together with a significant proportion of valid behavior – recall is reduced by more than 40% (4) Outlier detection rules are able to filter on average 43% of erroneous traces while only wrongly filter 4% of valid ones.

Furthermore, experiments using simulation measuring precision and recall in the two dimensions of increasing percentage of error (i.e. robustness) and increasing model size (i.e. scalability) of sk-strings and SMArTIC were performed. 1,800 experiments on three percentage of error levels and 160 experiments on different configurations of the number of nodes and the maximum number of transitions of specification model were performed.

From the robustness experiments, the precision of sk-strings is reduced proportionally to the error induced. On the other hand, only a slight reduction of precision is observed for SMArTIC. Our experiments also show the limitation of coring method in removing errors. From the scalability experiments, both sk-strings and SMArTIC are adversely affected by the increase in model scale (number of nodes). However, SMArTIC is able to retain better precision as compared to sk-strings up to factor of over 100.

Our experiments have strongly supported our belief that SMArTIC can produce more precise results with good recall and equivalent, or even better, co-emission in the presence
of errors and increasingly large model.

8. 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. We would also like to thank Glenn Ammons and Rastislav Bodik for sharing the detail of their coring algorithm. Last but not least, we would also like to thank Jianyong Wang and Jiawei Han for providing the executable of BIDE.

9. REFERENCES
10. APPENDIX

The following figures are procedures referred to by AutomatonMerge algorithm (outlined in Figure 11 at subsection 3.4).

**Procedure CreateUnifiableList**

**Inputs:**
X: First automaton
Y: Second automaton
td: top-down or bottom-up merge

**Outputs:**
Uni: Unifiable mapping of nodes in X and Y

**Method:**
Let nodesX = List of nodes by traversing X breadth first
Let nodesY = List of nodes by traversing Y breadth first
Let Uni = Empty If (!td) Reverse nodesX and nodesY
Let bChange = T
Let PrevDep = Empty
while (bChange) do
Let Dep = Empty
Let PairSet = PrevDep is Empty? X.Nodes × Y.Nodes:
- PrevDep
  For every (nx, ny) in PairSet do
    bChange = bChange || Call CheckUnifiable
    (nx, ny, Uni, Dep, td)
  end for
  bChange = bChange || Call SolveCircDep (Dep, Uni)
end while
Output Uni

Figure 17: Create Unifiable List

**Procedure CheckUnifiable**

**Inputs:**
nodeX: A node in automaton X
nodeY: A node in automaton Y
Uni: Unifiable nodes
Dep: Dependencies in CNF format

**Outputs:**
bUnifiable: Node Unifiable? T: F

**Method:**
Let txS = {}
Let tyS = {}
if (!td)
  Set txS = nodeX.Next
  Set tyS = nodeY.Next
else
  Set txS = nodeX.Prev
  Set tyS = nodeY.Prev
end if
Set bUnifiable = bUnifiable || Call GenCons(txS, tyS, Uni, Dep)
if (LabelRepeatExist(txS) || LabelRepeatExist(tyS))
  Set bUnifiable = bUnifiable &&
  Call GenCons(tyS, txS, Uni, Dep)
end if
if else bUnifiable = F
if (bUnifiable) Add (nodeX, nodeY) to Uni
Output bUnifiable

Figure 18: Check Unifiable

**Procedure GenCons**

**Inputs:**
TransToCheckX: X transitions
TransToCheckY: Y transitions
Uni: Unified list of nodes

**Outputs:**
bUnifiable: Node Unifiable? T: F

**Method:**
Let AND = Empty set of AND-ed constraints
for every tx from TransToCheckX
  Let OR = Empty set of OR-ed constraints
  for every ty from TransToCheckY
    if (tx.Lbl == ty.Lbl && AND.Unifiable(tx.Sink, ty.Sink))
      OR.Add ((tx.Sink, ty.Sink))
    end if
  end for
  AND.Add (OR)
end for
for Dep.Add ((tx.Source, ty.Source), AND)
if (bUnifiable) Add (#AND == 0?) T: F
Output bUnifiable

Figure 19: Generate Constraint
Procedure SolveCircDep

Inputs:
Dep : Dependencies in CNF format
Uni : Unifiable node list

Outputs:
bExist: Circular dep. exists? T:F

Method:
Let bExist = F
for (i=0;i<#Dep;i++)
  Let Pair = Dep.GetDep(i).Pair
  Let AND = Dep.GetDep(i).AND
  for at most #Dep times do
    if (∀ OR ∈ AND, (OR.Contains(Pair) || ∃ cons ∈ OR, Uni.Unifiable (cons)))
      bExist = T
    Uni.Add(Pair)
  end if
  Let NxtAND = Empty AND-ed dependencies
  for each OR ∈ AND
    if (OR.Contains(Pair))
      NxtAND.Add(OR)
    end if
  end for
  Let NxtOR = Empty OR-ed dependencies
  for each cons in OR
    Let NxtCons = Dep.Get(cons)
    NxtOR.Add(NxtCons)
  end for
  Let NxtOrCNF = Convert NxtOr to CNF
  NxtAND.AddRange(W/o Duplicate(NxtOrCNF)
  end for
Set AND = NxtAND
end for

Output bExist

Figure 20: Solve Circular Dependency

Procedure CreateStructuralMerge

Inputs:
XStart : First automaton (X) start node
YStart : Second automaton (Y) start node
MStart : Merged automaton start node
WX : Weight of automaton X
WY : Weight of automaton Y
Uni : Unifiable nodes
XUsed,YUsed: Nodes used for merging (so far)
TransXUsed,TransYUsed: Merged trans. (so far)

Method:
Let TransToCheckX = XStart.Next
Let TransToCheckY = YStart.Next
for each tx in TransToCheckX
  Let nX = tx Sink
  Let bestScore = 0
  Let tBest = Null
  for each ty in TransToCheckY
    Let nY = ty Sink
    if (Uni.Unifiable (nX,nY))
      Let PotSet = {(tx,ty) ∈ nX.Next × nY.Next| tx.Label==ty.Label ∧
                    Uni.Unifiable(tx.Sink,ty.Sink)}
      Let score = |PotSet|
      Update bestScore & tBest
    end if
  end for
  if (bestMatch != NULL)
    TransXUsed.Add (tx)
    TransYUsed.Add (tBest)
    Let m = XUsed.Get (tx.Sink)
    Let bRecurse = F
    if (m == NULL)
      Set m = Create new node
      XUsed.Add (tx.Sink,m)
      YUsed.Add (tBest.Sink,m)
      Set bRecurse = T
    end if
    Let dWX = WX/(WY + WX)
    Let dWY = WY/(WY + WX)
    Let prob = dWX * tx.Prob + dWY * tBest.Prob
    MStart.Next.Add (m,tx.Label,prob)
    if (bRecurse) Call CreateStructuralMerge
      (tx.Sink,tBest.Sink, m, WY, MStart, XUsed, YUsed, TransXUsed, TransYUsed)
    end if
  end if
end for
end for

Figure 21: Create Structural Merge
Procedure CreateEqClass

Inputs:
Merge: Mergable nodes
RepObjSet: Representative objects

Outputs:
DS: Data Structure

Method:
Step 1: Create equivalence class
Let EC = Empty
for each nX in (node | (node, ny) ∈ Merge) do
    Let equivY = {node | (nX, node) ∈ Merge}
    Set equiv = equivY ∪ {oX | (oX, ny) ∈ Merge ∧ ny ∈ equivY}
    Let repObj = Create new node
    RepObjSet.Add (repObj)
    EC.Add (repObj, equiv)
end for

Step 2: Add transitions among rep. objects
for each (repObj, equiv) in EC do
    Let node = first node in equiv
    for each t in node.Next do
        Let nextN = t.Sink
        Let nextRO = EC.GetRepObj (nextN)
    end for
end for

Step 3: Create Data Structure
Let DS = Empty
for each node ∈ Merge ∪ Merge do
    Let DescMap = {descendant, dRep | descendant is reachable from node ∧ dRep = EC.GetRepObj(descendant)}
    Let TransSet = trans — trans is traversable from node
    Let aDS = Create a data structure
    Set aDS.RepObj = EC.GetRepObj (node)
    Set aDS.DescMap = DescMap
    Set aDS.TransSet = TransSet
    DS.Add (aDS)
end for

Output DS

Figure 22: Create Equivalence Class

Procedure CreateFullMerge

Inputs:
node: A node
DSSet: Set of Data Structure
TVS: List of node traversed before
SHR: Shared nodes in structural merge
W: Weight
NUsed, TransUsed: Merged nodes and trans. of X where node ∈ X

Method:
if (node ∈ TVS) return
else TVS.Add (node)
Let mx = NUsed.Get (node)
Let mx = SHR/W:1
for each tx in node.Next do
    Let nextX = tx.Sink
    if (tx ∈ TransUsed) do
        if (∃ (nextX, d) ∈ NUsed)
            NUsed.AddRange (aDS.DescMap)
            TransUsed.AddRange (aDS.TransSet)
        else
            Let nextRep = Create new node
            mx.Next.Add (nextRep, tx.Lbl, mx.prob)
            NUsed.Add (nextX, nextRep)
            TransUsed.Add (tx)
        end if
    else
        Let nextRep = NUsed.Get (nextX)
        if (∃ t ∈ mx.Next | t.Sink = nextRep ∧ tx.Lbl = t.Lbl)
            t.AddProb (mw * tx.Prob)
        else
            NUsed.Add (nextX, nextRep)
            TransUsed.Add (tx)
        end if
    end if
end if
Call CreateFullMerge (nextX, DSSet, TVS, SHR, W, NUsed, TransUsed)
end for

Figure 23: Create Full Merge