SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner

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, compounded by imprecise, and changing requirements, along with the "short time to market" requirement, commonly leads to a lack of up-to-date specifications. This can result in software that is characterized by bugs, anomalies and even security threats. Software specification mining is a new technique to address this concern by inferring specifications automatically. In this paper, we propose a novel API 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 traces should be pruned from the input traces to a miner, and (2) Clustering related traces will localize inaccuracies and reduce over-generalizationin learning. Correspondingly, SMArTIC comprises four components: an erroneous-trace filtering block, a related-trace clustering block, a learner, and a merger. We show through experiments that the quality of specification mining 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—*Temporal API Specification Mining*

General Terms

Algorithms, Design, Reliability, Experimentation

Keywords

Clustering Traces, Filtering Errors, Specification Mining

1. INTRODUCTION

Improper management of software evolution commonly leads to a lack of up-to-date specifications (*cf.* [8]). This situation is further aggravated by imprecise, changing requirements, along with the "short time to market" requirement [5]. Low coherence between specifications and implementations can result in software that are characterized by 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 approaches to address these problems, especially in the area of specification discovery [6, 12, 34, 1, 11, 39, 19, 3]. These methods are generally termed *specification miners*. In [14], Fox illuminates the use of machine learning to bridge the gap between high level abstractions expressing software engineering problems and low level program behaviors. He points out that some baseline models can be learned automatically to aid in the characterization and monitoring of systems.

Specification miners can be classified into two groups, depending on how the mined specifications are represented: *automaton-based* [1, 6, 39, 34, 3] and *non-automaton based* [19, 11, 12, 33] specification miners.

The work by Ammons *et al.* is a pioneer in automaton-based specification mining [1]. There, a machine-learning approach is employed to discover program specifications 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 specifications can be obtained by their technique. Specifically, their technique focuses on mining of specifications which reflect temporal and data dependency relations of a program through traces of its API-client interaction. The specifications discovered model API-client interaction protocols, which are expressed initially as a probabilistic finite state automaton (PFSA). To reduce the effect of errors in training traces, transitions with a 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 the mining result for two primary reasons:

1. Early identification and filtering of erroneous program

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

execution traces can improve the quality of specification discovery.

2. Over-generalization which occurs at the 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 (*eg.*, "`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 [18].

We conduct experiments to support our reasoning. Our experiments aim at deriving the API interaction protocol for a client of the Jakarta Commons Net open-source library [28]. We adopt an objective measurement of the quality of our mining architecture through introduction of *accuracy*, *robustness* and *scalability*.

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

These measurements extend from the existing set of measurements found in the literature; specifically, the measurement of *accuracy* is supported by three kinds of metrics: precision, recall and co-emission. Measurement of *recall* and *precision* are suggested by Nimmer and Ernst [30]. They have been used as a measurement of soundness and completeness of specification discovery. As our specification miner manipulates specifications in PFSA format, it is necessary to also measure if the mined PFSA generates *the same traces at similar frequencies*, and thus places emphasis on similar sub-protocols. This is commonly measured by a metric known as *co-emission* [26].

The outline of the paper is as follows: In section 2, we provide a brief introduction to issues pertaining to experimentation on our specification miners. Section 3 lays down the hypotheses which drive our construction of SMArTIC, and discusses the detail components in SMArTIC. Section 4 describes our experiments on the Jakarta Commons Net [28] open source library. Section 5 describes more comprehensive experiments and results using various simulated models. We discuss related work in Section 6 and conclude in Section 7.

2. PRELIMINARY

In order to assess the quality of SMArTIC, we need to have an objective means of measuring an output automaton against an expected automaton. To this ends, we devise the following simulation model to facilitate experimentation with a variety of automata: We begin with a known API-interaction specification of a software component. This specification enables us to generate a set of simulated traces. Each trace is simply a sequence of API method invocation information. We set up a client to take in the simulated traces, and issue the corresponding method calls to interact with the component. 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.

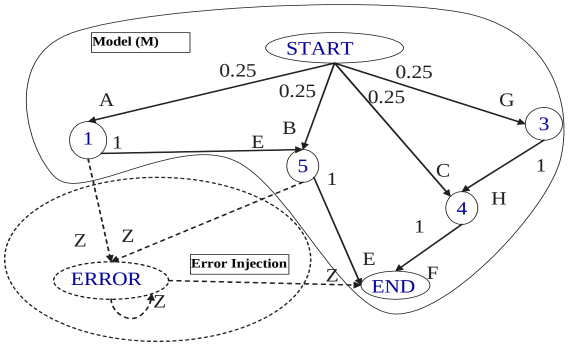

Figure 1: Sample Simulator Model

Our simulation model provides a controlled environment for experiments (*cf.*, [24]). Given a software component under examination, we can alter the input API-interaction specification and later compare it with the corresponding output specification. This preliminary section provides a summary of our work on a quality assurance framework for *automaton-based* specification miners [9].

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 generate *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 (drawn as dashed lines in Figure 1) model invalid API method calls from the state represented by its source node. This inclusion of error nodes and error transitions enable generation of erroneous traces; it aids the evaluation of the 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 a variety of model sizes different levels 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.

An actual program trace can be mapped to a string over an alphabet as shown by Ammons *et al.* through a "standardization" process [1]. Strings of symbols generated by our specification model (*aka.*, simulated traces) can be considered as an abstraction of actual program execution traces; *i.e.* a symbol representing a particular method call. Based on this abstraction, we generate simulated traces as strings of symbols.

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 sourcing 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 a *stratified random walk* guided by the probability of PFSA's transitions [27]. 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 a 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 [15, 31]. Given a PFSA $\mathcal{M}$ 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 $\mathcal{M}$ 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 details of our trace generation algorithm can be found in [9].

2.3 Precision, Recall and Co-emission

The terms *precision* and *recall* 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 [37].

Analogously, traces can be considered as documents and automata as a pool/population of documents. Let the original automata be denoted by X and the inferred automata by Y. Precision and recall can then be defined as *the proportion of traces in Y that is accepted by X* and *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 to address this probabilistic concern.

Co-emission has been used in measuring similarity between two Hidden Markov Models. Lyngsø *et al.* propose several versions of similarity measurements [26]. One such metric which is adopted here, denoted by PS, provides an unbiased and normalized similarity measurement of two models X and Y:

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

Here, $P_{CE}(X,Y)$ denotes a *co-emission probability*, determining the probability that a sentence s is generated by both X and Y independently. It measures how similar are the probabilities assigned to traces accepted by both X and Y. $P_X(s)$ 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 areas (*eg.*, [1], [12], [22], *etc.*). The success of SMArTIC hinges on the affirmation of the following two hypotheses:

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

HYPOTHESIS 2. *Mined specifications will be more accurate when they are obtained by merging the specifications learned from clusters of related traces than when they are 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 the output automaton, erroneous transitions are included during mining. Consequently, the 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 negate the effect of learning.

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

Hypothesis 2 is derived from the observation that the existence of unrelated traces may negate the effect of learning via generalization; *ie.*, 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.

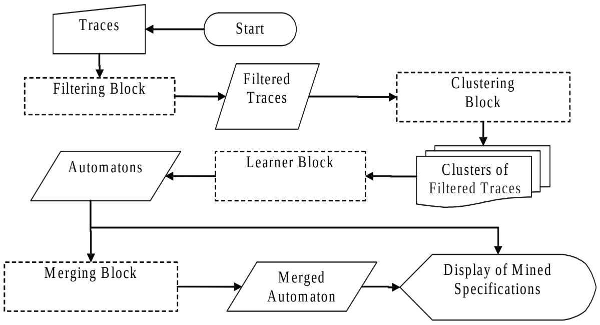

Figure 2: SMArTIC Structure

The overall structure of SMArTIC is as shown in Figure 2. It comprises 4 major blocks, namely filtering, clustering, learning and merging blocks. Each block is in turn composed of several major elements. The filtering block filters erroneous traces to address the robustness issue. The clustering block divides traces into groups of "similar" traces to address scalability issue. The learning block generates specifications in the form of automata. The merging block merges the automatons generated from each cluster into a unified one.

3.1 Filtering Block

The 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 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, rules of the form $\texttt{pre} \rightarrow \texttt{post}$, where both pre and post are sequences of alphabets occurring in traces. Semantically, such a rule has the following temporal interpretation: Given $\texttt{pre} = a_1 \ldots a_m$ and $\texttt{post} = b_1 \ldots b_n$, the temporal interpretation of $\texttt{pre} \rightarrow \texttt{post}$ is expressed in Linear Temporal Logic (LTL) notation [17] as

$$G(XF(a_1 \rightarrow \ldots \rightarrow XF(a_m \rightarrow XF(b_1 \wedge \ldots \wedge XF\ b_n))))$$

As an example, a rule $a \rightarrow bc$ asserts that 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* [16]). Support of a rule $\texttt{pre}{\rightarrow}\texttt{post}$ is the number of *trace points* exhibiting the property $\texttt{pre} \rightarrow \texttt{post}$. Confidence of the rule is the ratio of the number of *trace points* exhibiting the property $\texttt{pre} \rightarrow \texttt{post}$ to those exhibiting the property 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 $\texttt{pre}{\rightarrow}\texttt{post}$ format and exhibiting the above properties *outlier detection rules*.

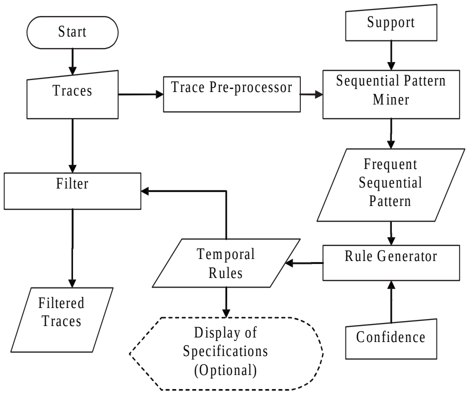

Figure 3: Filtering Block Structure

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

$$\begin{array}{l} \exists G(\texttt{pre} \rightarrow \texttt{post}) \in RS. \\ \quad (\exists a_i, a_j.\ (1 \leq i \leq j) \wedge \neg(a_i \ldots a_{j-1} \text{ satisfies } \texttt{pre}) \\ \qquad\qquad \wedge(a_i \ldots a_j \text{ satisfies } \texttt{pre}) \\ \qquad\qquad \wedge\neg(a_{j+1} \ldots a_{end} \text{ satisfies } \texttt{post})) \end{array}$$

Implementation-wise, the structure of the filtering block is as shown in Figure 3. Outlier detection rules can be extracted efficiently by adding pre and post processing steps to a closed sequential pattern miner, BIDE [38]. The end result of the filtering block is a multi-set of filtered traces. The algorithmic details can be found in our technical report [25].

3.2 Clustering Block

Input traces might be "mixed up" from several unrelated scenarios, *e.g.* a 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.

The 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 metric; *ie.*, measurement of similarity between two traces. The performance of the clustering algorithm is affected by appropriate similarity/distance metric. Different clustering algorithms learn differently in terms of accuracy, efficiency and the level of user interaction required. (*c.f.* [16]) The general structure of the clustering block is as shown in Figure 4.

Clustering Algorithm We use a classical off-the-shelf clustering algorithm for our purpose, namely the k-medoid algorithm [21].[1] The k-medoid algorithm works by computing

[1] Another algorithm is K-means algorithm. It is not used here since we would be required to define the average/mean of a group of strings which might not be meaningful. Also, k-

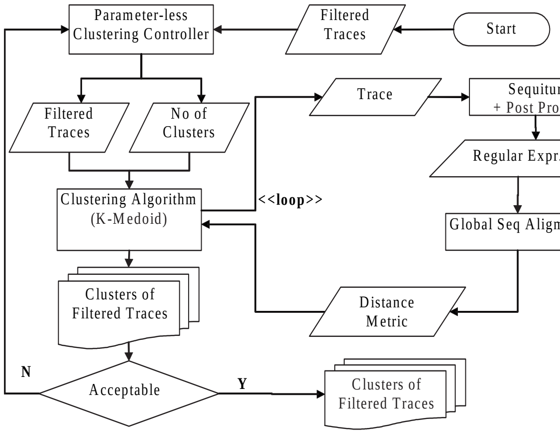

Figure 4: Clustering Block Structure

the distance between pairs of data items based on a similarity metric; this corresponds to computing the distance between pairs of traces. It then groups the traces with small distances apart into the same cluster. The 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.* [13] 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 a local maximum is reached. A detailed discussion of the implementation is available in our technical report [25].

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

Our first idea is to use *global sequence alignment* [36] 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 the Knuth-Morris-Prat (KMP) algorithm [7], the sequence alignment algorithm finds the best approximated alignment(s) rather than the occurrence of an 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 characteristics which make 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, consider the following program segment:

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

Suppose that APICLIENT_ABCD is a client function of an API. It is conceivable that the API interaction patterns for various runs via the 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 (only parentheses and "+" quantifier are used) 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.

We obtain the regular expression representation by converting a trace to its hierarchical grammar representation using Sequitur [29]. The output of sequitur will be post-processed to construct the regular expression representation and then be fed in as input to global sequence alignment. With these we obtain a method to find a reasonable distance metric for the similarity of program traces. We refer the readers to our technical report [25] for implementation details.

3.3 Learning Block

Although temporal rules have also been used ([41]) 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 automatons 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 [32].

Sk-strings learner is used by Ammons *et al.* to mine the specification of the X11 windowing library [1]. It is an extension of the k-tails heuristic algorithm of Biermann and Feldman [4] for learning stochastic automata. In k-tails, two nodes in a constructed automata are checked for equivalence by looking at 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

medoid algorithm has been found to be more accurate than k-means since it is less susceptible to outliers [16].

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 from them.

3.4 Merging Block

The merging process aims to merge multiple PFSAs 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 as the combined set (*i.e., union*) of sentences accepted by the multiple PFSAs.

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 PFSAs, thus improving scalability.[2]

Merging of two PFSAs involves identification of *equivalent* transitions between the two PFSAs. The output PFSA contains all transitions available in the input PFSAs, with each set of equivalent transitions represented by a single transition. Two identically-labelled transitions from two different PFSAs are considered *equivalent* if one the following conditions holds: (1) Both their source nodes share the same set of suffixes, (2) Both their sink nodes share the same set of prefixes.

Given a node n in a PFSA and a string accepted by PFSA that involves a transition, δ say, emitted from n, we define a suffix of n with respect to the string as the suffix of that string beginning with δ. Similarly, we define a prefix of n as the prefix of that string ended just before δ. For instance, given the transition $n_1 \xrightarrow{\delta} n_2$, and a string

$$t_1 t_2 \cdots t_{m-1} \delta t_{m+1} \cdots t_p,$$

A suffix of n_1 is the string $\delta t_{m+1} \cdots t_p$, and a prefix of n_1 is the string $t_1 t_2 \cdots t_{m-1}$.

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 prefixes/ suffixes of n in a PFSA is the set of prefixes/suffixes of n with respect to all strings accepted by the PFSA.

The definition of equivalent transitions above admits closure property: If two transitions are equivalent as defined by sharing the same set of suffixes, then each of the transitions in the suffix-set is also an equivalent transition. The same behavior can be observed from equivalent transitions sharing the same set of prefixes.

The merging process also ensures that the likelihood of traces generated by the output PFSA remains the same as that of the combined input PFSAs. More specifically, let $\mathcal{A}$ be the output PFSA and $\mathcal{A}_i$ $(i = 1..n)$ be the input PFSAs. Let δ 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 ℓ located in PFSA $\mathcal{A}$ and its equivalent transitions occurring in $\mathcal{A}_i$. $p_{\mathcal{A}_i}(\ell)$ is assigned to 0 if no equivalent transitions occur 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.

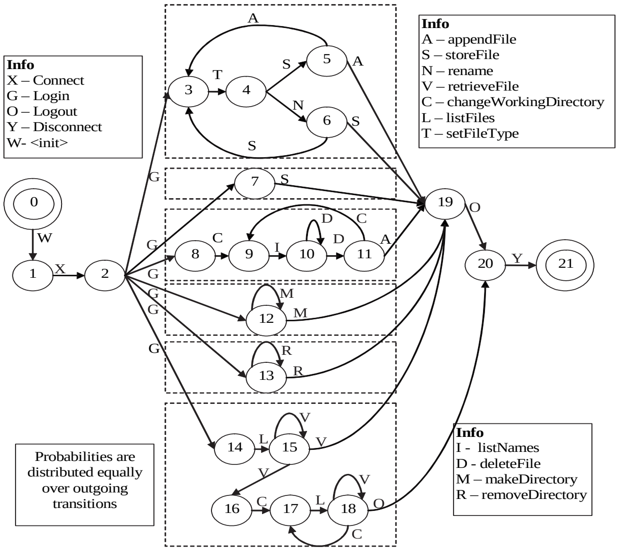

Figure 5: CVS Protocol

Implementation-wise, the closure property of equivalent transitions enables an incremental detection of such transitions, starting from either the start node (for finding prefixes) or the end node (for finding suffixes) of a PFSA. The detailed merging process is described in our technical report.

4. JAKARTA COMMONS NET

Jakarta Commons Net [28] 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 an 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 with a certain protocol pattern. Our CVS class and Commons Net library can be instrumented to generate traces which were then inputted to SMArTIC and sk-strings. The resultant models are then compared with the original CVS specification to evaluate the feasibility of SMArTIC 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 5. 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

[2] A secondary objective is to enable comparison of the structural similarity between the mined PFSA and the original PFSA. We omit the detail in this paper.

runtime analysis toolkit [20]. 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. The following features were added:

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. The class constructor might call other methods. We would like to capture the information that their context are the same but different from method calls called at constructor context.

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

The result of JRat instrumentation is an injection of tracing byte code into java classes. Running the instrumented code will produce a tree of method calls capturing their order and context represented as an XML document. Methods called earlier will print earlier in the XML document, each method will have its context as its ancestors in the XML tree.

4.3 Trace Collection and processing

We construct a wrapper class that takes in the automata shown in Figure 5 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 an exception happens we simply logout and disconnect from the FTP server. This is simulated by adding error transitions as shown in Figure 6. 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.

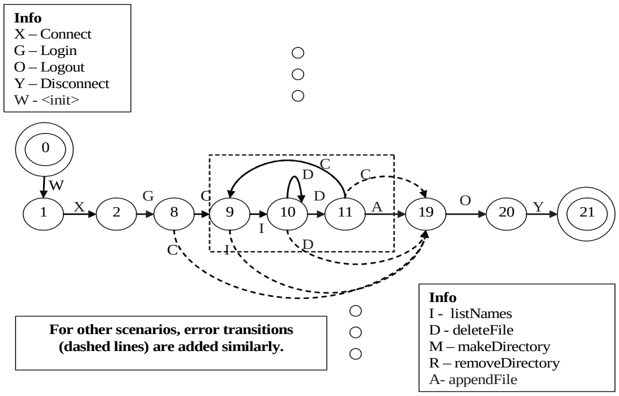

Figure 6: CVS Protocol With Error

4.4 Protocol Specification Generation and Results

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

A protocol specification is then produced and compared against the original one (as shown in Figure 5) 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).

	Precs	Recall	PS
SMArTIC	0.484	0.981	0.653
SMArTIC w/o filtering	0.426	1	0.616
SMArTIC w/o clustering	0.263	0.984	0.532
sk-strings(coring)	0.289	0.581	0.447
sk-strings	0.225	1.000	0.533

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

Both filtering and clustering help in increasing precision while maintaining good recall and equivalent or even better co-emission.

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 unavoidably 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. The results show that significant improvement (with at least 95% confidence level) in SMArTIC over the Sk-strings. Interested readers may refer to our technical report [25] for detail.

5. FURTHER EXPERIMENTS

The experiment with CVS specification in Section 4 provides positive evidence that SMArTIC 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 SMArTIC, 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 SMArTIC.

Our experiments use the same set of miners, with sk-strings learner being employed either in standalone mode or in co-operation with other processes, especially as the learner block of SMArTIC.

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 SMArTIC. For SMArTIC case, we disable the clustering sub-system to measure the effect of the filtering block.

We simulated the automaton generated by Ammons *et al.* in their analysis of the X11 windowing library (*cf.* [1]) – as shown in Figure 7. However, we modified the model slightly so that it was *without any non-determinism and repeated 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 8(a) & (b). Each of the two types of injections of errors shown in Figures 8(a) and (b) respectively corresponds to a separate set of sub-experiments.

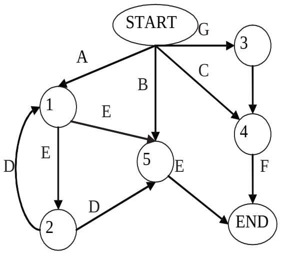

Figure 7: X11 Windowing Library Model

We expect the specification miners to be able to filter out errors. We compared the inferred automaton with the simulator model shown in Figures 8(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 percent levels 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 is 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.

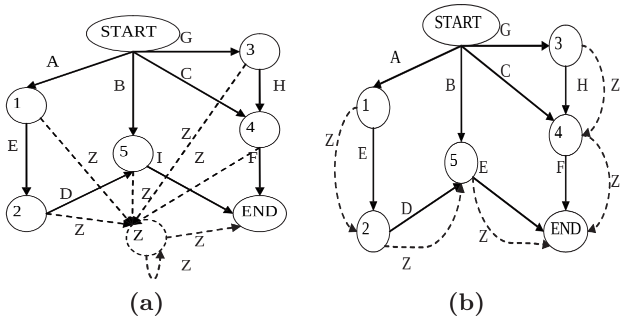

Figure 8: Models of Specification with Error

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 a maximum of m transitions per state with common start and end nodes. Transition labels were chosen from a pool of a 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). Details of our model generation algorithm can be found in our technical report [25].

These experiments were performed on sk-strings (standalone) and SMArTIC. In the SMArTIC case, we measure the effect of the clustering block by disabling the filtering sub-system.

We generated traces and capped the maximum number 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 a model of large size. This was not the case in our experiments, as only once was the cap reached; for the other 159 experiments, the 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 injections of errors. The models with different injection of errors are shown in Figures 8(a) and (b).

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

Error Model 1							
sk-strings				**sk-strings(coring)**			
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.946	1.000	0.946	4	0.999	0.823	0.864
8	0.908	1.000	0.948	8	0.998	0.828	0.867
10	0.883	1.000	0.950	10	0.990	0.845	0.875
SMArTIC							
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.999	1.000	0.946	10	0.981	1.000	0.948
8	0.993	1.000	0.946				
Error Model 2							
sk-strings				**sk-strings(coring)**			
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.947	1.000	0.947	4	0.829	0.962	0.863
8	0.898	1.000	0.947	8	0.812	0.909	0.848
10	0.875	1.000	0.948	10	0.816	0.886	0.849
SMArTIC							
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.994	1.000	0.946	10	0.974	1.000	0.949
8	0.986	1.000	0.946				

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.

The SMArTIC result is similar to sk-strings in that all traces generated by the simulator model X was accepted by the inferred model Y. Different from sk-strings, we noted only a slight drop in the acceptance of traces generated by Y. This drop is far less than the noise injected ((0.1%,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. The 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 a single transition, a set of transitions, a single path or a set of paths). In the second sub-experiment, erroneous transitions *go to valid nodes* instead of the 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 the maximum transitions per node at 4. In the second set of sub-experiments, we varied the maximum number of transitions while keeping the total number of nodes constant at 10.

Results. The experiment results are shown below for sk-strings and SMArTIC. Columns X.N and X.TN correspond to the number of nodes and maximum number of transitions per node in the specification model.

Varying No of Nodes						
sk-strings				**SMArTIC**		
X.N/X.TN	Precs	Recall	PS	Precs	Recall	PS
10/4	0.437	1.000	0.560	0.584	0.988	0.690
20/4	0.003	1.000	0.015	0.185	0.998	0.227
30/4	0.0004	1.000	0.005	0.059	0.999	0.090
40/4	0.0004	1.000	0.004	0.014	1.000	0.007
Varying Max No.Of Transitions						
sk-strings				**SMArTIC**		
X.N/X.TN	Precs	Recall	PS	Precs	Recall	PS
10/3	0.113	1.000	0.218	0.453	0.984	0.504
10/5	0.187	1.000	0.284	0.578	0.993	0.424
10/7	0.084	1.000	0.213	0.500	0.984	0.508
10/9	0.073	0.997	0.087	0.514	0.990	0.329

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 (*ie.*, 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 the maximum number of transitions increased.

6. RELATED WORK

There have been numerous work in the area of specification mining. They can be classified into two groups, depending on how the mined specifications are represented: *automaton-based* [1, 6, 39, 34, 3] and non-automaton based [41, 11, 12, 33] specification mining.

The specification miner described in [1] has been extensively referenced in this paper. In other work, Whaley *et al.* extract object-oriented component interface sequencing constraints to form multiple finite state automatons [39]. Reiss *et al.* encode program execution traces as directed acylic graphs to aid visualization and understanding of programs [34]. Arts *et al.* dynamically extract program models from Erlang programs as state graph models for model checking and visualization [3]. We believe that these and other similar miners can benefit from our architecture with minimal changes.

Separating Error and Normal Traces. Weimer and Necula contrasted (previously labelled) error and normal traces to statically mine and filter temporal specifications for error detection in the form of 2 event rules [40]. In our case, we use dynamic analysis to mine automaton-based specification, also unsupervised learning is used to identify and filter erroneous traces. In [2], Ammons *et al.* utilizes hierarchical filtering to aid a specification miner user to detect and delete clusters of error traces *en masse*. Their method heavily depends on user input while the method proposed in this paper is automatic.

Mining Rules for Software Engineering Tasks. In [11], association rules (*ie.*, describing association of items rather than the sequence of events) based on frequent itemset mining have been used to describe user interaction behavior for web-based GUIs.

In [41], Yang *et al.* present their work in mining a restricted form of *response* pattern [10] called an "alternating" pattern using a set of templates. They only consider rules involving two events; *ie.*, 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 a partial solution rule set for multi-event (> 2) sequential patterns.

SMArTIC also mines a temporal *response* pattern [10] 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 a *chain response* pattern in [10]. Moreover, rather than mining all possible statistically significant rules, we only mine those useful for outlier detection.

Program Trace Characterization. Larus proposes using whole program path (WPP) to represent a program's dynamic control flow [23]. Sequitur [29] is used to compact acyclic program traces into WPP which is 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 [35]. Sequitur [29] 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 traces. 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 the grammar output of sequitur directly, we post-process it into a regular expression (regex) format. The regex representation "flattens" the grammar while retaining its structure. Two regex representations can then be easily compared by global sequence alignment which produces their best alignment.

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 clusters 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-)protocol. 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 rule 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 groups useful for filtering. They can effectively capture those property patterns proposed in [10] which are interesting for program traces and useful for identifying potential bugs.

We experimented with the Jakarta Commons Net open-source library. Our experiments aim at deriving API interaction protocols for client applications of Jakarta Commons Net open-source library [28]. 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.

Further 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 tests on three percentage of error levels and 160 tests on different configurations of the number of nodes and the maximum number of transitions of the 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 the 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 the anonymous referees for their valuable comments. We would also like to thank Anand Raman, Peter Andreae and Jon D. Patrick for letting us use their sk-strings learner in our experiment, and Glenn Ammons and Rastislav Bodik for sharing the detail of their coring algorithm. Furthermore, we thank Jianyong Wang and Jiawei Han for providing the executable of BIDE, and Hugh Anderson for his help in proof-reading this paper.

9. 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.R. Larus. Debugging temporal specifications with concept

analysis. In *Proc. of Programming Language Design and Implementation*, 2003.

[3] T. Arts and L. Fredlund. Trace analysis of Erlang program. In *Proc. of Erlang Workshop*, 2002.

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

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

[6] 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.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C.Stein. *Introduction to Algorithms.* MIT Press, 2001.

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

[9] D.Lo and S-C.Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proc. of Working Conf. on Reverse Engineering*, 2006.

[10] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of Int. Conf. on Software Engineering*, 1999.

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

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

[13] A. Foss and O.R. Zaiane. A parameterless method for efficiently discovering clusters of arbitrary shape in large datasets. In *Proc. IEEE Int. Conf. on Data Mining*, 2002.

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

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

[16] J. Han and M. Kamber. *Data Mining Concepts and Techniques.* Morgan Kaufmann, 2001.

[17] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge, 2004.

[18] The Java hotspot performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[19] J.Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proc. of the Euro. Conf. of Object Oriented Programming*, 2003.

[20] JRat. http://jrat.sourceforge.net/.

[21] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis.* John Wiley & Sons, 1990.

[22] E. Keogh, S. Lonardi, and C.A. Ratanamahatana. Towards parameter-free data mining. *Proc. of Int. Conf. on Knowledge Discovery and Data Mining*, 2004.

[23] J.R. Larus. Whole program paths. In *Proc. of Programming Language Design and Implementation*, 1999.

[24] A.M. Law and W. D. Kelton. *Simulation modelling and analysis.* McGraw-Hill, 2000.

[25] D. Lo and S-C. Khoo. SMArTIC: Specification mining architecture with trace filtering and clustering. In *SoC-NUS tech. report, TRA 8/06*, 2006.

[26] R.B. Lyngsø, C.N.S. Pedersen, and H. Nielsen. Metrics and similarity measures for hidden Markov models. In *Proc. of National Conf. on Artificial Intelligence*, 1999.

[27] D.S. Moore and G.P. McCabe. *Introduction to the Practice of Statistics.* W.H. Freeman & Co., Third Edition, 1998.

[28] Jakarta Commons Net. http://jakarta.apache.org/commons/net/.

[29] C.G. Nevill-Manning, I.H. Witten, and D.L. Maulsby. Compression by induction of hierarchical grammars. In *Proc. of Data Compression Conf.*, 1994.

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

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

[32] 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.

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

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

[35] S.P. Reiss and M. Renieris. Generating java trace data. In *Proc. of the ACM Conf. on Java Grande*, 2000.

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

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

[38] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. of Int. Conf. on Data Engineering*, 2004.

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

[40] W.Weimer and G.Necula. Mining temporal specifications for error detection. In *Proc. of Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[41] 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.