Automatic Steering of Behavioral Model Inference*

David Lo
School of Information Systems
Singapore Management University
{davidlo}@smu.edu.sg

Leonardo Mariani and Mauro Pezzè
Dept. of Informatics Systems and Communication
University of Milano Bicocca
{mariani,pezze}@disco.unimib.it

ABSTRACT

Many testing and analysis techniques use finite state models to validate and verify the quality of software systems. Since the specification of such models is complex and time-consuming, researchers defined several techniques to extract finite state models from code and traces. Automatically generating models requires much less effort than designing them, and thus eases the verification and validation of large software systems. However, when models are inferred automatically, the precision of the mining process is critical. Behavioral models mined with imprecise processes can include many spurious behaviors, and can thus compromise the results of testing and analysis techniques that use those models.

In this paper, we increase the precision of automata inferred from execution traces, by leveraging two learning techniques. We first mine execution traces to infer statistically significant temporal properties that capture relations between non consecutive and possibly distant events. We then incrementally refine a simple initial automaton by merging likely equivalent states. We identify equivalent states by analyzing set of consecutive events, and we use the inferred temporal properties to evaluate whether two equivalent states can be merged or not. We merge equivalent states only if the merging does violate any temporal property, since a merging that violates temporal properties is likely to introduce an imprecise generalization. Our generalization process that preserves temporal properties while merging states avoids breaking non-local relations, and thus solves one of the major cause of overgeneralized models. Thus, mined properties *steer* the learning of behavioral models. The technique is completely automated and generates an automaton that both accepts the input traces and satisfies the mined temporal properties.

We evaluated our solution by comparing models inferred with and without checking mined temporal properties. Results show that our steering process can significantly improve precision without noticeable loss of recall.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

Algorithms, Verification

Keywords

Mining Automata, Dynamic Analysis, Temporal Properties

1. INTRODUCTION

Monitoring applications at testing time and in-the-field produces useful information that describes the behavior of software systems. We can use traces to automatically infer baseline models that can help software engineers understand, verify and validate software systems [11]. For example, we can trace inter-component method calls, and use the recorded traces to derive models that summarize and generalize interactions between components [25].

Techniques that extract and generate behavioral models can be classified into *automaton* (for example, [3, 7, 17, 23, 31, 34]) and *non-automaton based* (for example, [9, 8, 20, 35]) techniques, depending on the nature of the models that they generate. Techniques of both kinds generate models that can support test case generation [13, 24], debugging [30, 25] and verification [26]. In this paper, we focus on automaton based techniques, namely techniques that generate finite state automata (FSA) from execution traces.

Classic learning algorithms infer FSA that are neither precise nor complete: Mined models can include many spurious behaviors that can hinder the effectiveness of downstream analysis techniques. In a recent study, Lo et al. show that automata learners generate imprecise and overgeneralized models especially when models are large and complex [16]. Overgeneralized models produce many false negatives or positives that significantly reduce effectiveness of debugging, verification and validation techniques.

In this paper, we present a technique that generates precise FSA by using temporal properties inferred from traces to steer the generation of FSA. Classic algorithms that infer FSA generate an initial tree that matches the structure of the traces (prefix tree acceptor), and incrementally merge

*This work has been partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

ESEC/FSE '09 Amsterdam, The Netherlands

states that are equivalent according to their local behaviors. For instance, some popular algorithms merge states with outgoing transitions that generate the same sequences up to a given length. Such strategies refine models without considering the non-local effect introduced by merging states. For example, inference algorithms with a limited look-ahead can merge the intermediate events of the sequences <openFile, doX, ..., closeFile> and <openConn, doX, ..., closeConn>, and generate imprecise behavioral models that accept spurious sequences like <openFile, doX, ..., closeConn> and <openConn, doX, ..., closeFile>.

We avoid such problems of imprecision by steering the inference of FSA with temporal properties that are inferred from traces, and that capture relations between potentially distant events. In the simple example drafted above, we can easily infer temporal properties that relate openFile with closeFile events, and openConn with closeConn. We then use these properties to prevent merging states that violate the inferred properties, and thus we do not generate the spurious sequences produced by classic inference algorithms. The final generated automaton will both satisfy all the temporal properties that are initially identified and will accept all the traces provided as input. This method is completely automated, but can also accept user input in the form of a set of temporal properties that must be satisfied by the resulting inferred automata.

We developed a framework similar to the one presented in [35, 20, 19] to automatically discover temporal properties in the form of future and past-time temporal rules. Depending on the complexity of the analyzed traces, we can tune complexity of inferred temporal properties to scale to large traces. For instance, the algorithm for inferring temporal properties of length less or equal 2 is less complex than the algorithm for inferring the FSA, and has no impact on the complexity of the overall approach, while the algorithm for inferring temporal properties of unbounded length is more complex than the algorithm for inferring the FSA, and has a significant impact on the complexity of the overall approach.

We empirically compared the learning of FSA with and without steering with a set of automatically generated automata and case studies that describe the behavior of software systems. Our evaluation shows that inferring automata with temporal properties significantly improves precision without loss of recall. Moreover, the overhead introduced by the steering is extremely limited: in our experiments the cost of steering has always been between 4% and 12% of the learning phase.

The paper is organized as follows. Section 2 presents our framework to infer automata with steering. Section 3 presents a technique to infer temporal rules. Section 4 describes the *kTail* algorithm to infer regular automata. Sections 5 extends the *kTail* algorithm with efficient incremental refinement based on temporal rules. Section 6 describes the empirical experience with our solution and presents qualitative and quantitative results. Section 7 discusses related work, and Section 8 concludes.

2. THE MINING FRAMEWORK

The framework to infer automata presented in this paper works in two phases: (1) rule mining and (2) automata learning with steering. In the rule mining phase, we process the execution traces to learn a set of statistically significant temporal rules. In the automata learning phase, we generate automata, using the mined rules to prevent imprecise generalizations. Figure 1 shows the relation between the two main phases. In the following, we discuss the rationale underlying the two phases.

Mining rules. We infer temporal properties structured as pre- and post-condition pairs, where the post-condition is expected to hold if the corresponding pre-condition is true. We mine rules to steer the inference of automata, and thus we need to identify properties that hold with perfect confidence and high support. A temporal property holds with perfect confidence (100%) if every time a pre-condition holds, the corresponding post-condition holds as well. Lower confidence levels identify properties that hold in many, but not all cases. A rule has a high support if its pre-condition is observed a large number of times. It is desirable to mine rules with high support to prevent inference of incidental properties. A rule with high confidence but low support could correspond to an exceptional case rather than a general rule.

Temporal properties inferred with perfect confidence and high support represent relevant constraints that hold in the target program. Temporal properties capture relations between events independently from their distance in the traces, and thus include relations between non-consecutive events that can be separated by long sequences of intermediate events in the trace files. Popular algorithms for learning automata often miss relations between events that are far in the traces, and can thus result in over-generalizations that may introduce many behaviors extraneous to the mined software. The technique proposed in this paper avoids undesired over-generalizations by enforcing temporal properties mined from traces while inferring the automata. As shown later, this can significantly increase the precision of the learned automata.

The set of mined temporal rules can be refined by users, i.e., users can modify, add and delete rules. User intervention is not necessary for the technique and in most cases we envision our tool to be used without any manual intervention. However, if users are aware of some rules that must or must not hold, manual refinement of the mined rules is supported by our solution.

Learning automata with steering. Many automata learners have been integrated in testing and analysis techniques to mine behavioral models [31, 25, 17, 23]. One of the first and most popular inference algorithm is the *kTail* algorithm proposed by Biermann and Feldman in [4]. Many variants of this algorithm have been investigated [29, 7]. All these algorithms are based on a common schema: First build a prefix tree acceptor (PTA) that represents the input traces, and then incrementally merge equivalent states until obtaining the final automaton. A prefix tree acceptor is a tree representation of a set of traces. Each trace corresponds to a path in the tree. Traces that share the same prefix, share also a sub-path in the tree. States are merged if they share their *next future*, that is, based on similarities among the outgoing transitions up to a given distance from the states. As mentioned above, this can lead to imprecisions, when states share their next future, but differ in their distant future. We augment classic automata learners with mined rules to prevent merging that leads to imprecise automata: We merge nodes only if the automata produced by the merging satisfies all the mined rules. We refer to this process as automata

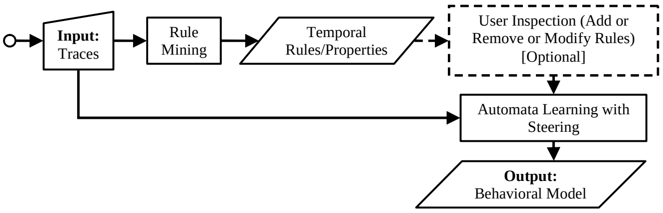


Figure 1: Deriving precise automata using mined rules

No	Trace
1	$\langle UP, W, X, G, T, S, A, O, Y \rangle$
2	$\langle DEL, W, X, G, C, L, D, O, Y \rangle$

Table 1: Sample traces obtained by monitoring events in a CVS client application.

mining *with steering*, since the rules "steer" the inference of the automata. In this paper we illustrate the steering process on the well known kTail algorithm.

3. MINING TEMPORAL RULES

In this section, we define events, traces and temporal properties, and we provide additional details about the framework for mining temporal properties that we integrated in our solution.

Let I be a set of distinct events where an event corresponds to a behavior of interest, for instance, a method call. A trace corresponds to an ordered sequence of events in I. Formally, each trace is a tuple $\langle e_1, e_2, \ldots, e_{end} \rangle$ where $(\forall_{1 \leq i \leq end}.\ e_i \in I)$. The input to our mining framework is a set of traces.

There are many types of temporal constraints that can be inferred from traces. In this work, we focus on future-time and past-time relations, which can capture many relevant properties that are common in software systems.

We define a rule R as a pair of sequences of events that correspond to the pre- and the post-condition of the rule, respectively. A *future-time* temporal rule is denoted as *pre* $\rightarrow$ *post*, where *pre* and *post* are the pre- and post-condition, respectively, and $\rightarrow$ denotes the future-time implication operator. A *past-time* temporal rule is denoted as *pre* $\hookrightarrow_P$ *post*, where *pre* and *post* are the pre- and post-condition, respectively, and $\hookrightarrow_P$ denotes the implication operator.

Future-time temporal rules specify relations of the type "Whenever a series of events happened (pre-condition), another series of events must eventually happen (post-condition)". Techniques for mining future-time temporal rules have been defined by Yang et al. and Lo et al. [35, 20]. Past-time temporal rules specify relations of the type "Whenever a series of events happened (pre-condition), another series of events must have happened before (post-condition)". Techniques for mining past-time temporal rules have been introduced by Lo et al. [19].

To infer temporal rules efficiently, we can limit the length of the rules to be inferred, where the length of a rule is the sum of the length of the pre-condition (that is the number of events occurring in the pre-condition) and the length of the post-condition (that is number of events occurring in the post-condition). We use temporal properties to steer the learning of automata, thus, we need properties that hold with high probability. We measure the probability that properties hold by computing the statistical metrics of support and confidence, commonly used in data mining [12]. The support of a rule is the number of times its pre-condition appears in the traces. The confidence of a rule is the likelihood that the pre-condition is always followed by the post-condition. In our experiments, we used only rules with a support greater or equal 20% and confidence of 100%, that is rules whose pre-condition appears $\geq$ (20% $\times$ the number of traces), and is always followed by the post-condition in all traces.

Our solution can mine rules with or without limiting the length of the mined rules. When mining rules of unlimited length, we can derive complex rules, but we strongly impact on the performance of the solution. When mining rules of limited length, we derive simple rules, but we keep the inference time short. We found that limiting the length of the mined rules to 2 can be extremely effective, and scales well to large programs and models. Studying the advantage and disadvantages of mining longer rules is part of future work. In this paper we consider mining rules of length 2.

The algorithm for mining future and past time rules of length 2 works as follows:

1. Mine all frequent pre-condition events of length one with support $\geq$ 20%.
2. For each of the pre-conditions do the following:
   - Find all occurrences of the pre-condition.
   - Find all frequent post-conditions of length one, scanning the traces forward (future time) and backward (past time)
   - Report all rules with a confidence of 100%

The algorithm is a variant of the algorithms presented by Lo et al. [20, 19] that infer rules of unlimited length.

The above algorithm for mining rules of length 2 has a complexity of O(n + ($a * b$)), where n is the cumulative length of all the traces, b is the total number of occurences of all frequent events (with support of at least 20%), and b is the maximum length of a trace. We can improve time performance by allocating a quadratic space. However, the main bottleneck of the approach is the automata learning, which has a complexity of $O(n^2$ x $|A|^k)$, where n is the cumulative length of all the traces, $|A|$ is the size of the alphabet and k is the length of the tail considered when evaluating equivalence between states.

Table 2 shows an example of the results of the algorithm applied to the set of traces shown in Table 1 that are obtained by monitoring a CVS client: The algorithm discov-

Type	Rule
Future	$\langle DEL \rangle \rightarrow \langle D \rangle$
Future	$\langle UP \rangle \rightarrow \langle S \rangle$
Future	$\langle W \rangle \rightarrow \langle X \rangle$
	...
Past	$\langle D \rangle \hookrightarrow_P \langle DEL \rangle$
Past	$\langle S \rangle \hookrightarrow_P \langle UP \rangle$
Past	$\langle G \rangle \hookrightarrow_P \langle W \rangle$
	...

Table 2: Rules mined from traces in Table 1.

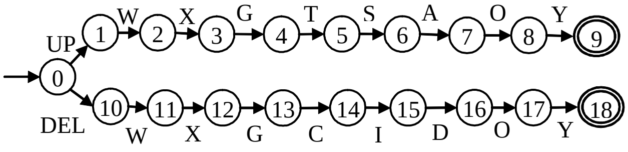
ered a total of 40 past-time rules and 44 future-time temporal rules of length 2 with a support level greater or equal to 20% and confidence level of 100%.

4. LEARNING AUTOMATA

In this section, we present the *kTail* algorithm that we use to learn automata. The *kTail* algorithm automatically infers FSA by building an initial tree that represents the input traces, and then progressively refining the representation by merging equivalent states to obtain the final FSA.

The initial tree is a prefix tree acceptor (PTA), where each input trace maps to a branch in the PTA. The language accepted by the tree is the set of initial traces. Figure 2 shows a sample PTA built from the two traces in Table 1. The traces have been extracted from executing a Concurrent Versions System (CVS) client built upon Jakarta Commons FTP library described in [17].

The labels UP and DEL indicate the uploading and deleting of files to and from the CVS server, respectively. The symbols W, X and G indicate the initialization of the FTP library, the `connect` and the `login` commands, respectively. The symbols T, S and A indicate the execution of `setFileType`, `storeFile` and `appendFile` commands, respectively. The symbols C, I and D indicate the execution of `changeWorkingDirectory`, `listNames` and `deleteFile` commands, respectively. Finally, the symbols O and Y indicate the execution of `logout` and `disconnect` commands, respectively.



Legend
Double circles represent final states.

Figure 2: Sample Prefix Tree Acceptor (PTA)

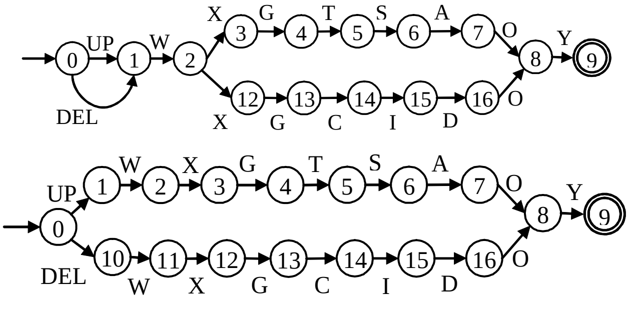
The *kTail* algorithm transforms the initial PTA into a FSA by merging states that are equivalent according to some equivalence criterion. Various algorithms use different equivalence criteria depending on the desired degree of generalization [4, 29, 25]. The *kTail* algorithm merges states whose equivalence is computed by comparing their tails of length k, where the tails of length k are the sets of event sequences that exit the states and include at most k events[1]. More formally, an automaton A is defined as $A = (\Sigma, S, s0, \delta, F)$, where:

- Σ is the input alphabet (a finite, non-empty set of symbols).
- S is a finite, non-empty set of states.
- $s_0 \in S$ is an initial state.
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$ (here for simplicity we assume that the automaton is deterministic, but the definition can be generalized to non-deterministic automata).
- $F \subseteq S$ is the set of final states.

Given a sequence of symbols $w = \langle e_1, \ldots e_n \rangle$, the extended state transition function $\delta^*(s, w)$ is defined as $\delta^*(\delta(s, e_1), \langle e_2, \ldots e_n \rangle)$ if $w \neq \epsilon$, and s otherwise.

Given a state s, its future of length k (k-future) is the set of sequences of maximum length k that can be accepted by δ^* when applied to state s.

Two states are k-equivalent if they have the same k-future. The k-equivalence relation captures the intuitive idea that if two states are not distinguishable by looking to their near future, they probably represent the same conceptual state, and thus can be merged. In the practical cases, to mine a general enough model from limited traces, the value of k must be small. It is usually fixed between 2 and 4 [31, 7, 25]. As a result, two states are often merged even if they are not close to be equivalent, thus implicitly introducing many undesired behaviors in the FSA model. For example, the top part of Figure 3 shows the automaton resulting from the execution of kTail on the PTA in Figure 2 with $k = 2$. We can notice that kTail merges the states 1 and 10, and 2 and 11, and thus produces a FSA that accepts many erroneous traces.



Legend
Double circles represent final states.

Figure 3: FSA Built Using kTail (top) and kTail with Steering (bottom)

5. INFERENCE WITH STEERING

This section describes how we use temporal properties to steer the learning of automata with improved precision. The

[1] We do not require that a sequence ends with a final state, following the description in [7].

```
Procedure ViolateMinedProperties
Inputs:
$FSA$: A partial FSA
$S1$: The first candidate state to be merged
$S2$: The second candidate state to be merged
$FRules$: The set of future-time temporal rules
$PRules$: The set of past-time temporal rules
Outputs:
True if the FSA violates any temporal property, false otherwise.
Methods:
1:  For each $fr \in FRules$
2:      Check for an occurrence of $fr$.pre in $FSA$ which is not followed by an $fr$.post before $S1$. If
        the check holds, check if there exists a sequence of events from $S2$ to an end node in the in
        $FSA$ which which does not contain $fr$.post. If the latter check succeeds, return true.
3:      Check for an occurrence of $fr$.pre in $FSA$ which is not followed by an $fr$.post before $S2$. If
        the check holds, check if there exists a sequence of events from $S1$ to an end node in the in
        $FSA$ which which does not contain $fr$.post. If the latter check succeeds, return true.
4:  For each $pr \in PRules$
5:      Check for an occurrence of $pr$.pre in $FSA$ which is not preceded by an $pr$.post after $S1$. If
        the check holds, check if there exists a sequence of events from $S2$ to a start node in $FSA$
        which does not contain $pr$.post. If the latter check succeeds, return true.
6:      Check for an occurrence of $pr$.pre in $FSA$ which is not preceded by an $pr$.post after $S2$. If
        the check holds, check if there exists a sequence of events from $S1$ to a start node in $FSA$
        which does not contain $pr$.post. If the latter check succeeds, return true.
7.  return false
```

Figure 4: A sound and complete algorithm for detecting imprecise merging

initialization step of the *kTail* algorithm produces a PTA that accepts all the input traces and satisfies all the temporal rules by construction. In fact, since each branch that terminates in a final state corresponds to a trace, and each mined rule has a perfect confidence level, the language accepted by the PTA includes the exact set of traces, and satisfies the set of inferred rules.

The *kTail* algorithm incrementally merges k-equivalent states until no merges are possible. In order to refine the model without violating the inferred temporal properties, we augment the classic *kTail* merging step that compares k-futures, with checks that verify wether the FSA resulting from the merging satisfies the inferred temporal properties or not, and we allow the merging of k-equivalent states only if it preserves the temporal properties.

In general checking LTL properties over FSA (model checking) is NP-hard [6], and this is the complexity of our general configuration. However, since we limit the length of the mined rules to 2, we can check our LTL properties in linear time with respect to the size of the model.

We defined two algorithms to check whether temporal properties of length 2 are satisfied by a refined FSA or not. The first algorithm is *sound* and *complete*, but requires many passes through the model. the second algorithm traverses the model only once, and is thus faster than the first algorithm, it is *sound* but *incomplete*. We chose the algorithm depending on the tradeoff between performance and the required level of accuracy.

Figure 4 shows the *sound* and *complete* algorithm that detects imprecisions when merging two k-equivalent states $S1$ and $S2$. For each future-time temporal rule, the algorithm checks whether its pre-condition occurs before $S1$ and the post-condition is neither satisfied before $S1$ nor after $S2$ (line 2). If it is true, the FSA resulting from the merge violates the future-time temporal rule and thus we do not merge $S1$ with $S2$. The algorithm checks each past-time temporal rule similarly (line 4). Since the algorithm checks for future- and past-time temporal rules of length 2, it can verify each rule by traversing each state of the FSA at most 2 times. If all the temporal properties hold, the algorithm for detecting imprecise merges returns false and the two states are merged.

Since we can have many rules and the model can be large, traversing the entire model can be expensive. Thus, we defined a second algorithm for detecting imprecise merges that can check each rule in constant time, at the price of incompleteness. The algorithm is *sound*, since it detects only true violations of the temporal rules, but *incomplete*, since it may miss some violation, and thus can accept FSA that violates some temporal rules. Figure 5 shows the fast algorithm.

The algorithm computes the set of events that can occur before and after nodes $S1$ and $S2$ in the FSA, and stores the computed sets in a hashtable (lines 1-4). The algorithm checks all temporal rules in constant time by using the set of events stored in the previous step, and thus avoiding scanning the sequences for each rule. The fast algorithm assumes that a temporal rule is satisfied by the past/future of a state if both the pre- and the post-condition of the rule are included in the pre-computed past and future sets, thus it may miss some property violations, and incorrectly merge the states.

A future time temporal rule fr is violated if its pre-condition $fr.pre$ holds before occurrence of state $S1$ (thus the merge of $S1$ and $S2$ can potentially affect fr) and its post-condition $fr.post$ holds neither before $S1$ nor after $S2$. Line 6 in Figure 5 formally specifies these conditions. A similar check that inverts the role of $S1$ and $S2$, detects violations of a past time temporal rule as shown in line 8 of Figure 5.

Each past time temporal rule *pr* is violated if its pre-condition *pr.pre* holds in the future of $S1$ (thus the eventual merge of $S1$ and $S2$ can potentially affect *pr*) and its post-condition *pr.post* holds neither after $S1$ nor in the past of $S2$. Line 11 in Figure 5 formally specifies these conditions. A similar check is performed after inverting the role of $S1$

```
Procedure ViolateMinedPropertiesApprox
Inputs:
$FSA$: Partial FSA to be considered
$S1$: The first candidate state to be merged
$S2$: The second candidate state to be merged
$FRules$: The set of future-time temporal rules
$PRules$: The set of past-time temporal rules
Outputs:
True if the FSA violates any temporal property, false otherwise.
Methods:
1:  Let $S1.nxt$= Events occurring after S1
2:  Let $S1.prev$ = Events occurring before S1
3:  Let $S2.nxt$= Events occurring after S2
4:  Let $S2.prev$ = Events occurring before S2
5:  For each $fr \in FRules$
6:      If $fr.\text{pre} \in S1.prev \wedge fr.\text{post} \notin S1.prev \wedge fr.\text{post} \notin S2.nxt$
7:          return true;
8:      If $fr.\text{pre} \in S2.prev \wedge fr.\text{post} \notin S2.prev \wedge fr.\text{post} \notin S1.nxt$
9:          return true;
10: For each $pr \in PRules$
11:     If $pr.\text{pre} \in S1.nxt \wedge pr.\text{post} \notin S1.nxt \wedge pr.\text{post} \notin S2.prev$
12:         return true;
13:     If $pr.\text{pre} \in S2.nxt \wedge pr.\text{post} \notin S2.nxt \wedge pr.\text{post} \notin S1.prev$
14:         return true;
15: return false;
```

Figure 5: A fast, sound but incomplete algorithm for detecting imprecise merging

and $S2$, as shown in line 13 of Figure 5. If the candidate merge passes all the checks, the algorithm returns false and the states are merged.

The runtime complexity of the sound and complete algorithm for detecting imprecise merges is O($m \times r$), where m is the size of the model, and r is the number of rules. Since the model can be large and the number of rules can be high, the overall runtime costs can be high, especially considering that checks must be performed for each potential node merge. The runtime complexity of the fast algorithm for detecting imprecise merges is O($m + r$), and thus the fast algorithm scales well to large and complex cases.

We illustrate the benefits of the steering process with the simple example discussed in the previous section: the learning of a FSA from the two traces that have been produced by executing the CVS protocol, and are shown in Table 2. The algorithm presented in this paper can handle FSA of any complexity, including any combination of branches and loops. We chose a simple example to simplify the discussion.

Figure 3 (top) shows the FSA learned from the traces using kTail. Figure 3 (bottom) shows the FSA learned from the same traces using our steering approach with the mined rules shown in Table 2. The statistically significant mined rules $\langle D\rangle \hookrightarrow_P \langle DEL\rangle$ and $\langle S\rangle \hookrightarrow_P \langle UP\rangle$ prevent the merging of nodes 1 and 10 of the PTA shown in Figure 2, and thus do not produce an imprecise FSA. In fact, merging nodes 1 and 10 produces a FSA where an event UP can *potentially precede* an event D and an event DEL can *potentially preceded* an event S, hence violating the CVS protocol. In fact, a `delete` command (DEL) should not be followed by a `store` command (S), and an `upload` command (UP) should not be followed by a `delete` command (D). Both behaviors are accepted by the FSA learned with kTail, but are not accepted by the FSA learned with our steering approach.

The undesired generation of unexpected behaviors in inferred models is clear from this simple example. Imprecise merges could be more frequent when the structure of the inferred FSA is complex. In that case, many imprecise state merges can introduce a large amount of undesired behaviors in the mined model, thus hindering debugging, testing and analysis algorithms based on these models.

Superficially, it might seem that imprecision could be eliminated by simply increasing the value of k in the *kTail* algorithm to a large number. This is not the case, because even large values of k limit the merging decision to a set of events close to the considered states, while the steering approach proposed in this paper considers temporal properties that predicate on future events that may be very far from the compared states, and that may reflect semantically relevant properties of the software system. With very large values for k, states are merged only if they are almost fully equivalent, but the merging requires an almost complete set of observations to be effective. For example, with $k = 10$, the *kTail* algorithm merges states only if their outgoing behaviors of length ≤ 10 matches completely, but would merge almost no states, unless provided with an almost complete set of observations, and this occurs very seldom.

6. EMPIRICAL VALIDATION

We empirically evaluated the effectiveness of our algorithm by comparing the quality of the models learned using the *kTail* algorithm with the models learned using the *kTail* algorithm extended with our steering mechanism. We measured precision, recall and time required to perform the mining as evaluation criteria. Precision is the percentage of mined behaviors that are correct, namely the percentage of behaviors generated by the mined model that are accepted by the model to be inferred. Recall is the percentage of correct behaviors that have been mined, that is, the percentage of behaviors generated by the model to be inferred that are accepted by the mined model. We refer to [17, 16] for the definition and computation of precision and recall of mined FSA.

In our empirical validation, we considered three sets of empirical studies. In the first empirical study, we evaluated the effectiveness of our algorithm when mining real-life models. In the second empirical study, we evaluated our algorithm when addressing models of increasing size and complexity. In the third study, we evaluated effectiveness of our solution with traces that we generated by monitoring a real software system: the open source MP3 player jlGUI [15]. For the empirical validation we used an extended QUARK automata mining quality assurance framework [16] that automatically generates models from software executions, and traces from models, and that computes precision and recall for the mined models. In our empirical validation, we used rules of length 2 and the fast algorithm for detecting imprecise merges.

6.1 Effectiveness

We evaluated the effectiveness of our algorithm by experimenting with models that describe the behavior of three real applications: We generated traces from the models, we inferred models from traces by using both kTail and kTail with steering, and we compared the inferred models by computing precision and recall. We considered the X11 Windowing Library (studied in [3]), a CVS client application (studied in [17]) and the IBM® WebSphere® Business Integration processes from WebSphere® Commerce (studied in [36]). To collect a representative set of traces, namely a set of traces

System Model	|Evs.|	kTail			With Refinement		
		Precs.	Recall	Time	Precs.	Recall	Time
X11 Windowing Library	356.400	0.873	1.000	0.211	0.905	1.000	0.218
CVS Client	2121.000	0.169	0.970	0.557	1.000	0.970	0.616
WebSphere Business Processes	9317.080	1.000	0.999	1.453	1.000	0.999	1.528

Table 3: Empirical Results: Precision, Recall and Elaboration Time

that well represent the model, we instructed the QUARK framework to generate enough traces to cover each transition in the model at least 2 times.

We compared efficiency and effectiveness of kTail and kTail with steering by measuring the time necessary to generate the models, and by computing precision and recall for the resulting automata. Table 3 shows the results of our experiments. We computed the values of time, number of events in the input traces (|Evs|), precision, and recall by repeating each experiment 25 times with 25 different sets of traces, and by computing the average.

The results show that the precision of the models inferred with steering is significantly higher than the precision of the models inferred without steering. We observed the highest difference in precision when experimenting with the CVS client application. In this case, the precision of the inferred model raises from 0.169 in the case of models produced with kTail, to 1 in the case of models produced with steering. This impressive improvement is due to the characteristics of the CVS protocol that requires several properties between events that occur at the beginning and at the end of sequences of operations. Learning without steering generates overgeneralized models that do not satisfy these properties, while the steering mechanism successfully refines the inference by enabling only those generalizations that do not violate the relevant properties of the protocol. It is also worth noticing that in this study FSA inference with steering always generated models with almost perfect precision (2 out of the 3 models have perfect precision and 1 model has precision higher than 0.9). The data reported in Table 3 show that steering increases precision without any loss of recall.

The experiments confirm the expected limited cost of the steering mechanism: The extra time required for learning models with steering is always between $4-8\%$ of the overall mining time. Thus, that benefits in precision are obtained with limited overhead.

6.2 Scalability

To investigate the effect of our algorithm when addressing models of large size, we randomly generated models with an increasing number of states. To generate realistic combinations of simple and complex structures in each model, the model generator randomly assigned to each node from 0 to 3 edges.

In the empirical evaluation, we considered 4 configurations corresponding to models with 20, 30, 40 and 50 states. For each configuration, we generated 100 random models, we use each random model to generate traces, we fed the generated traces to both kTail and kTail with steering, and we computed precision and recall of the learned models with respect to the random model.

Table 4 shows results for the 4 configurations. Each value is computed by averaging the values obtained for the 100 random models. The results indicate an improvement in precision between 9% and 15%, which is far from the improvement experienced in the CVS example in Section 6.1. This is due to a limited number of activations of the steering process. Since models are randomly generated and little semantics is associated to event names, there are only a few rules that can be automatically discovered and used for the steering. The overhead introduced by steering is minimal (11% in the worst case).

To study the precision of the results, Table 5 shows precision and recall obtained by considering only the models where steering has been activated during the inference, which are the models that resulted in violations of some temporal rules. In this case, the improvement in precision is between 20.7% to 23.8%, while recall does not change significantly.

This study confirms that our algorithm can effectively manage models of different size and complexity. It also shows that our technique introduces stronger improvements with models that describe real software systems and protocols, where several relations between distant events exist, rather than random models, where few relations between events exist.

6.3 Experiments with jlGUI

We completed this early validation with an experiment on a real system: jlGUI, an opensource MP3 player [15] that represents the interesting case of a medium-size (11981 LOC) GUI-based application written in Java. We focused our analysis on the methods that are executed when managing playlists and selecting files to play. Table 6 reports the methods that we monitored during the experiments.

We instrumented the program using AspectJ, and we collected traces when executing different usage scenarios for the playlist management, with specific focus on adding songs. jlGUI provides two ways to add songs to playlists: through the playerUI button or the playlistUI button. In the former case, users can update the playlist by: (1a) deleting the current playlist and loading a song from a file, or (2a) uploading a saved playlist. In the latter case, users can extend the playlist by: (1b) appending one song to the current playlist, or (2b) appending all songs in a directory to the current playlist. We thus have 4 ways to add songs to playlists. Figure 6 shows the specification of these 4 cases that we derived manually. In this case, events represent executed methods.

We collected 15 traces by running the 4 scenarios with different combinations of data values. We learned FSA from the traces both with the kTail and the kTail with steering algorithms. kTail with steering learned the model perfectly with 100% precision and recall. kTail alone learned a model with 57.9% precision and 100% recall. This empirical study shows the benefit of our steering technique when analyzing a real software system.

7. RELATED WORK

There are many algorithms for learning behavioral models from execution traces, that are used to serve testing and analysis techniques. Here, we group learners into four main

Model Size	**|Evs.|**	**kTail**			**With Refinement**		
		Precs.	Recall	Time	Precs.	Recall	Time
20 Nodes	1644.200	0.715	0.996	0.252	0.801	0.996	0.272
30 Nodes	4026.870	0.456	0.998	0.339	0.609	0.997	0.367
40 Nodes	7476.180	0.377	0.999	0.458	0.496	0.999	0.509
50 Nodes	15560.090	0.251	0.999	1.302	0.361	0.999	1.416

Table 4: Empirical Results: Scalability vs. Accuracy

Model		**kTail**			**With Refinement**		
Size	Considered	Precs.	Recall	Time	Precs.	Recall	Time
20 Nodes	37	0.588	0.997	0.258	0.822	0.995	0.282
30 Nodes	72	0.400	0.997	0.354	0.611	0.997	0.382
40 Nodes	53	0.269	1.000	0.452	0.491	0.998	0.506
50 Nodes	53	0.197	0.999	1.249	0.404	0.998	1.357

Table 5: Empirical Results: Effective Steering Cases

classes: learning techniques augmented with steering mechanisms to learn FSA, techniques for mining models that describe the ordering of events, either based on FSA or not, and techniques for mining behavioral models not related to the ordering of events.

7.1 Learning FSA With Steering

To the best of our knowledge, the technique presented in this paper is the only completely automated approach that uses inferred temporal properties to steer the inference of FSA.

The learning technique that is closest to our approach is the one proposed by Walkinshaw and Bogdanov [33]. They augment the inference of FSA with a steering mechanism based on LTL properties. We apply the same principle, but with two main differences: (1) we infer the temporal properties automatically, while Walkinshaw and Bogdanov require user-specified properties for the steering process; and (2) we limit complexity of temporal properties to be checked, while Walkinshaw and Bogdanov extensively use the expression power of LTL. Requiring user inputs in the inference process can increase the precision of the temporal properties that are used for the steering process, but limits significantly the usability of the technique that requires the manual generation of many properties. Moreover the Walkinshaw and Bogdanov approach requires deep knowledge of the system, while our approach can be applied without any knowledge of the system. Finally, the extensive use of LTL properties can result in a more precise steering than the use of the simple temporal properties that we integrated in our solution. However, mining and checking complex LTL properties is expensive and does not scale well. Here we present an efficient technique of practical use that can generate many and large models with low execution overhead, at the cost of some loss in the expressiveness power of the temporal properties.

7.2 Learning FSA Without Steering

There are many algorithms for generating FSA from positive execution traces [2, 4, 17, 23, 25, 31]. Many approaches are variants of the popular *kTail* algorithm defined by Biermann and Feldman [4]. These techniques can produce useful models when the models to be inferred are small and simple, but when the size and the complexity of the models increase, the precision of the inferred models decrease dramatically [16].

Our solution extends the existing approaches to learn FSA by introducing an automated refinement method based on the preliminary mining of statistically significant temporal properties that are used to steer the FSA learning. Early results reported in this paper show that our solution can significantly increase precision of inferred models without any loss of recall.

The work by Lo and Khoo [17] filters erroneous traces during mining. In this work, we address the orthogonal issue of preventing imprecise state merges assuming that the input traces are correct. The two approaches are complementary and can be merged to produce a robust and accurate framework to detect both erroneous traces and imprecise merging.

7.3 Learning Models of Ordered Events

Although FSA are a popular model, there are many technique to learn other models that describe the order of events, like sequence diagrams, frequent pattern and temporal properties. In general, different models capture different aspects and are thus mutually complementary.

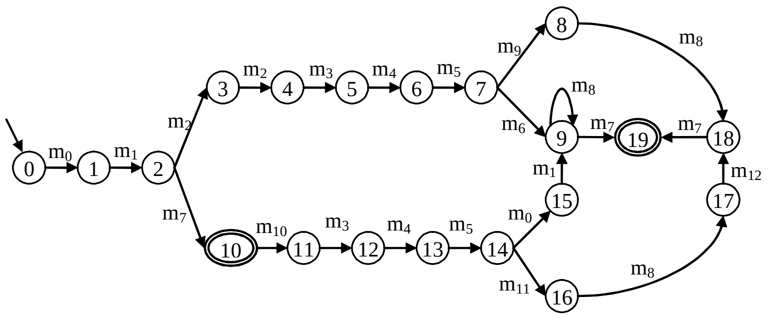
Many techniques for mining sequence diagrams simply represent traces as sequence diagrams [1, 14, 27]. Some of these techniques can also infer loops [5] or modalities [21, 22]. Different from our work, none of the above mentioned techniques offers a systematic mechanism for refining mined models using temporal properties.

Some techniques can infer patterns or sequences of events that repeat frequently in a set of traces [8, 18, 32]. In our work, we produce FSA as a final result. FSA can describe a more complete relations between events as compared to frequent patterns that only capture some linear relations between events that appear frequently. The more complete relations among events can be a very useful input to many testing and analysis techniques.

Finally, there are several techniques to mine temporal properties. Yang et al. present interesting work on mining two-event future-time temporal logic rules, which are significant with respect to a user-defined *satisfaction rate* [35]. Lo et al. extended the work by Yang et al. to mine significant future-time temporal rules of arbitrary lengths [20]. In further work, Lo et al. propose a method to mine significant past-time temporal rules of arbitrary lengths [19]. Differently from rules, automata can represent more relations between events than linear temporal rules. Inferred automata can include loops and branches, while temporal properties inferred with the frameworks proposed in [35, 20, 19] cannot capture disjunction in the mined temporal rules.

ID	Details
m_0	boolean javazoom.jlgui.player.amp.PlayerUI.loadPlaylist(String)
m_1	boolean javazoom.jlgui.player.amp.playlist.BasePlaylist.load(String)
m_2	void javazoom.jlgui.player.amp.playlist.ui.PlaylistUI.processActionEvent(ActionEvent)
m_3	File[] javazoom.jlgui.player.amp.util.FileSelector.selectFile(Loader, int, boolean, String, String, File)
m_4	File[] javazoom.jlgui.player.amp.util.FileSelector.selectFile(Loader, int, boolean, File, String, String, String, File)
m_5	void javazoom.jlgui.player.amp.util.Config.setLastDir(String)
m_6	void javazoom.jlgui.player.amp.playlist.ui.PlaylistUI.addDir(File)
m_7	void javazoom.jlgui.player.amp.PlayerUI.processActionEvent(ActionEvent)
m_8	void javazoom.jlgui.player.amp.playlist.BasePlaylist.appendItem(PlaylistItem)
m_9	void javazoom.jlgui.player.amp.playlist.ui.PlaylistUI.addFiles(File[])
m_{10}	void javazoom.jlgui.player.amp.PlayerUI.processEject(int)
m_{11}	void javazoom.jlgui.player.amp.playlist.BasePlaylist.removeAllItems()
m_{12}	void javazoom.jlgui.player.amp.playlist.BasePlaylist.nextCursor()

Table 6: Method calls traced in jlGUI



Legend
Double circles represent final states.

Figure 6: jlGUI Specification to be inferred

Our framework integrates a solution for mining the subset of the rules described in [20, 19] that can be scalably mined.

7.4 Learning Other Models

There are several techniques that learn behavioral models, not necessarily related to the ordering of event. The most popular example is Daikon, a mining approach to learn boolean expressions that describe likely invariants on values of program variables [9, 28, 10]. These models usually represent aspects that complement the information described by FSA.

8. CONCLUSIONS

Most techniques for mining FSA are based on generating an initial prefix tree acceptor that is heuristically and incrementally refined through a state merging process that ends when no states can be further merged. Studies reported in [16] shows that merges are frequently imprecise, and generate models with low precision, especially when models are large and complex.

To improve precision of the inference of FSA from traces, in this paper we propose a steering mechanism to refine the state merging strategy. Our technique is based on the inference of statistically significant temporal properties from traces. These properties are then used to steer the FSA inference process to prevent state merging that would produce automata that violate the inferred properties. The algorithm is designed to be scalable and to produce FSA that both generate all the input traces and satisfy the inferred temporal properties.

We evaluated our technique on a set of case studies that show how steering can significantly increase precision without loss of recall, and with a limited overhead on the inference process. Our studies also show that our solution can effectively manage real world models and scale with the size of applications.

9. REFERENCES

[1] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/, visited in 2008.

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining Specification. In *proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

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

[5] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] 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.

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

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

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

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

[12] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[13] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *proceedings of the 25th International Conference on Software Engineering*, pages 60–71, 2003.

[14] J. G. Hosking. Visualisation of object oriented program execution. In *proceedings of the IEEE Symposium on Visual Languages*, 1996.

[15] Javazoom. jlgui - music player. http://www.javazoom.net/jlgui/jlgui.html, 2009.

[16] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *proceedings of the 13th Working Conference on Reverse Engineering*, 2006.

[17] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2006.

[18] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.

[19] D. Lo, S.-C. Khoo, and C. Liu. Mining past-time temporal rules from execution traces. In *proceedings of the 6th International Workshop on Dynamic Analysis*, 2008.

[20] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[21] D. Lo, S. Maoz, and S.-C. Khoo. Mining Modal Scenario-Based Specification from Execution Traces of Reactive Systems. In *proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering*, 2007.

[22] D. Lo and S.Maoz. Mining scenario-based triggers and effects. In *proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[23] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *proceedings of the International Conference on Software Engineering*, 2008.

[24] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *proceedings of the 29th International Conference on Software Engineering*, 2007.

[25] L. Mariani and M. Pezzè. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.

[26] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, 2003.

[27] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (VET): an interactive plugin-based visualisation tool. In *proceedings of the 7th Australasian User Interface Conference*. Australian Computer Society, Inc., 2006.

[28] J. Perkins and M. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2004.

[29] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *proceedings of the Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

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

[31] S. P. Reiss and M. Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, 2001.

[32] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *proceedings of the International Conference on Program Comprehension*, 2006.

[33] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

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

[35] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *proceedings of the International Conference on Software Engineering*, 2006.

[36] Y.Zou, T. Lau, K. Kontogiannis, T. Tong, and R. McKegney. Model-driven businees process recovery. In *proceedings of the 11th Working Conference on Reverse Engineering*, 2004.