Size and Accuracy in Model Inference

Nimrod Busany, Shahar Maoz, Yehonatan Yulazari
Tel Aviv University
Tel Aviv, Israel

***Abstract*—Many works infer finite-state models from execution logs. Large models are more accurate but also more difficult to present and understand. Small models are easier to present and understand but are less accurate.**

In this work we investigate the tradeoff between model size and accuracy in the context of the classic k-Tails model inference algorithm. First, we define mk-Tails, a generalization of k-Tails from one to many parameters, which enables fine-grained control over the tradeoff. Second, we extend mk-Tails with a reduction based on past-equivalence, which effectively reduces the size of the model without decreasing its accuracy.

We implemented our work and evaluated its performance and effectiveness on real-world logs as well as on models and generated logs from the literature.

I. Introduction

Many works infer finite-state models from execution logs. The inferred models have different potential uses, from program comprehension and malware detection to finding problems in service levels, to list a few. Large models, with many states and transitions, are more accurate but also more difficult to present, analyze, and understand. Small models are easier to present, analyze, and understand, but are less accurate.

In this work we investigate the tradeoff between model size and accuracy in the context of the classic k-Tails model inference algorithm. k-Tails, introduced in [6], takes an execution log and a positive integer k as input, and constructs a finite-state machine (FSM), whose states correspond to unique sub-sequences of length k or less from the log, and which accepts a language that over-approximates the set of traces in the log. Roughly, the higher the parameter k, the more accurate the produced model. At the same time, however, the higher the parameter k, the larger the produced model. Over the last two decades, k-Tails was very popular and has been used, in many variants, by many different authors.

We investigate the tradeoff between model size and accuracy in two ways. First, we present mk-Tails, a generalization of k-Tails from a single parameter k to a set of parameters $\{k_1, \ldots, k_t\}$. Second, we present an efficient reduction in the size of the inferred model that does not affect accuracy.

Specifically, first, we observe that some elements in the logs, e.g., some events or sequences of events, may be of more interest than others. For example, based on domain-knowledge in security, the engineer may know that some events or sequences of events are more sensitive than others and require more accurate inspection. As another example, in the context of checking that a bug was fixed, the engineer may be more interested in events that were involved in the bug at hand than in other events that appear in the log. However, the k-Tails algorithm provides a single, fixed level of abstraction (over-approximation) for the entire log. In other words, the engineer cannot get a model that has less abstraction around some events and more around others; her control over the tradeoff between abstraction and model size is very limited.

Our new algorithm, mk-Tails, extends the k-Tails algorithm to accept a set of input parameters $\{k_1, k_2, \ldots k_t\}$, each of which applies to a subset of events in the log's alphabet, as selected by the engineer. Accordingly, mk-Tails produces a model whose level of abstraction varies; it is more accurate around events whose corresponding k is high and is less accurate around events whose corresponding k is low. This enables fine-grained control over the tradeoff. As we later show, increasing the value of k for the entire alphabet, as was done in all previous works, may result in a major increase in the model size. In contrast, our extension, mk-Tails, allows the engineer to make the model accurate where necessary and less accurate where it is not necessary, gaining more accuracy while paying less in model size, and thus better control the tradeoff.

Second, we observe that the model produced by k-Tails (or mk-Tails) may include many redundant states and transitions; the model is often larger than it needs to be. Thus, based on merging of states with equivalent past, we show a reduction in the model's states and transitions that preserves the model's language and thus reduces its size without reducing its accuracy. Importantly, unlike existing reduction algorithms for non-deterministic finite automata, our reduction is efficient, as it takes advantage of the information in the logs and the unique properties of the k-Tails algorithm, even before the model itself is constructed.

It is important to note that our present work does not make any new claims regarding the usefulness of the results of k-Tails for tasks for which it has been used in the past, program comprehension, test generation, log differencing, etc. Based on claims and evidence provided in previous works, by others, as we cite below, we assume that k-Tails results could be useful for software engineers, and focus solely on the challenges that are common to all applications of k-Tails: enabling better control over the tradeoff between model size and accuracy.

It is also important to note two additional assumptions underlying our present work. First, in this work we assume that smaller finite-state models are indeed easier to present and understand than larger finite-state models. Evaluating this assumption with engineers is outside the scope of our work. Second, the application of mk-Tails assumes that the engineer has domain knowledge or task-related knowledge that leads her to care a lot about accuracy around some events and care

less about accuracy around other events. Providing criteria for selecting these events of interest is domain-specific and task-specific, and is outside the scope of our present work.

We implemented our ideas and evaluated them over real-world logs as well as over models from the literature and logs that we generated from them. The results show that mk-Tails performs well and is able to produce models with high accuracy arround selected events, while not paying much in model size. Further, they show that our reduction scales well, and effectively reduces model sizes by 10%-40%. See Sect. VI.

Over the last two decades, k-Tails has been used, in many variants, in many works [2], [4], [5], [8]–[10], [12], [16]–[18], [20]–[23], [28], [31]. To the best of our knowledge, no work has considered a generalization of k-Tails to many parameters. In addition, no work has investigated applying reductions over the k-Tails model, while taking advantage of its unique structure and exploiting the data-structures from which it is inferred. See related work in Sect. VII-B.

II. Example

We use a small example to motivate and demonstrate our work, presented semi-formally, for illustration purposes. Formal definitions appear later in the paper. Consider a very small log, which consists of 5 traces, see Fig. 1. An engineer wants to produce a finite-state model that represents the behaviors that appear in the log. Ideally, the model should be informative, small, and accurate.

For a start, the engineer runs the classic k-Tails with $k = 1$. Fig. 2 shows the resulting model, which is a compact but a rather general one, perhaps too general. For example, the model accepts traces that contain the sequence `cd` $\rightarrow$ `lf` $\rightarrow$ `mkdr`, although this sequence is not part of any trace in the log.

The engineer may be happy with the size of the model but less happy with its spurious traces and the relatively high abstraction. Hence, she increases the value of k to reduce the abstraction in the model.

Fig. 3 shows the model produced by running k-Tails with $k = 2$. The new model no longer allows the spurious sequence `cd` $\rightarrow$ `lf` $\rightarrow$ `mkdr`. Indeed, the abstraction in this model has decreased dramatically. However, the size of the new model, when counting both states and transitions, grew by 35.3%, from 51 to 69, making it more difficult to comprehend.

Tr1	Tr2	Tr3	Tr4	Tr5
init	init	init	init	init
conn	conn	conn	conn	conn
in	in	in	in	in
sf	cd	cd	ln	ln
lf	lf	lf	rf	af
mkdr	sft	sft	del	rn
out	out	out	af	rf
dis	dis	dis	rn	del
clr	rm	clr	out	out
dis			reset	dis
			dis	
			clr	

Fig. 1: A log of 5 traces, inspired by the cvs.net model from [17]

To address this, the engineer uses our new algorithm, mk-Tails: as based on her knowledge of the domain, some events in the logs are more important than others, she uses mk-Tails to assign different ks to different subsets of the log's alphabet. Specifically, as an example, she uses $k = 2$ for the event `lf`, which based on her knowledge of the domain is more important than other events, and $k = 1$ for all other events.

Fig. 4 shows the output of mk-Tails in this case. Now, the model size, counting all states and transitions, has only grown from 51 to 52, yet it excludes the spurious sequence above. This demonstrates the ability of mk-Tails to deal with the tradeoff between model size and accuracy by fine-grained control using different abstraction levels around different events.

Finally, the engineer applies our past-equivalence reduction algorithm over the mk-Tails model. This reduces the size of the model without changing the language it accepts and thus without affecting its accuracy (and as we later show, is done efficiently over mk-Tails models).

Fig. 5 shows the result of applying the reduction to the model of Fig. 4. Note that states 4, 11, and 15 from Fig. 4 have been merged into state 22 from Fig. 5. Further, states 9 and 14 from Fig. 4 have been merged into state 23 from Fig. 5. The model now has 46 states and transitions, a reduction of 13%. Importantly, this final model accepts the same language as the mk-Tails model over which it was applied.

III. Preliminaries

We present basic definitions and background on k-Tails and NFA reductions, required for the later parts of the paper.

A. Basic Definitions

A trace over an alphabet Σ is a finite word $w = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle$, where $\sigma_1, \ldots, \sigma_m \in \Sigma$. For $j \geq 1$ we use w_j to denote the jth element in w. We use $|w|$ to denote the length of w. For a positive integer k, let $\Sigma_{\leq k}$ denote the set of all sequences of length k or less.

A log L over an alphabet Σ is a set of traces $L = \{w_1, \ldots, w_n\}$. We denote by $|L|$, $|L|_e$ the number of traces and events in the log resp.

Example 1. *For the log in Fig. 1,* Σ=`{sft, init, cd, conn, ln, in, af, lf, del, out, dis, sf, rf, rn, rm, mkdr, clr, reset}`, $|L| = 5$, $|L|_e = 50$.

Definition 1 (Finite-State Machine (FSM))**.** *A finite-state machine is a structure $M = \langle Q, I, F, \Sigma, \delta \rangle$, where Q is a set of states; $I \subseteq Q$ is a set of initial states; $F \subseteq Q$ is a set of terminal (accepting) states; Σ is an alphabet; and δ is a transition relation $\delta : Q \times \Sigma \times Q$.*

We use subscript notation to refer to the elements of an FSM model M. For example, δ_M refers to the transition relation of M. For $t = (q, \sigma, q') \in \delta$, t_s, t_σ, t_e denote q, σ, and q' resp.

Definition 2 (A Run)**.** *A run over an FSM model M, is a finite sequence of transitions that starts on an initial state and maps to transitions in M: $\langle t_1, t_2, \ldots, t_n \rangle$, s.t., $t_{1_s} \in$*

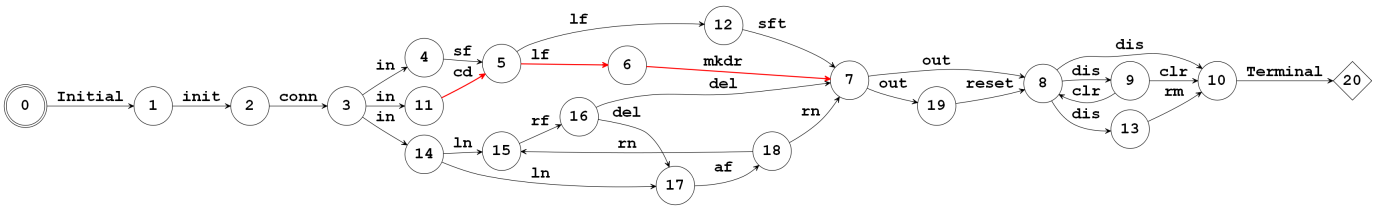

Fig. 2: The result of running k-Tails with $k = 1$ on our example log. The model has 21 states and 30 transitions, and contains the spurious sequence `cd`→`lf`→`mkdr` (red edges).

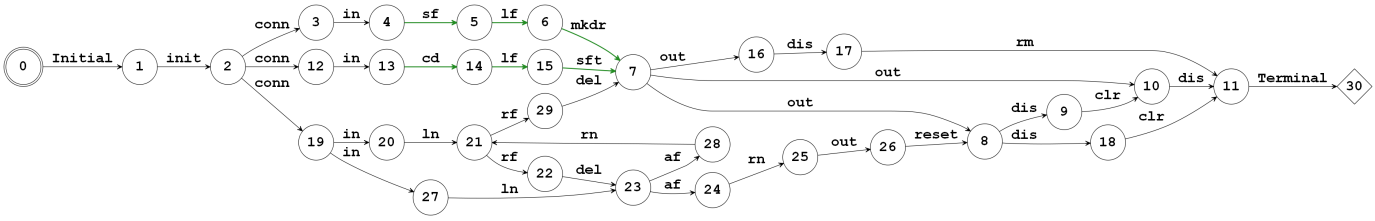

Fig. 3: The result of running k-Tails with $k = 2$ on our example log. The model has 31 states and 38 transitions.

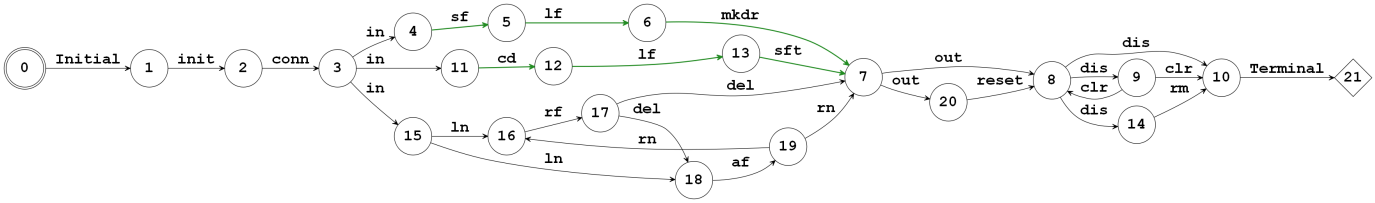

Fig. 4: The result of running mk-Tails with $k = 2$ for `lf`, and with $k = 1$ for all the other events in the log. The model has 22 states and 30 transitions. The spurious sequence `cd`→`lf`→`mkdr` is excluded.

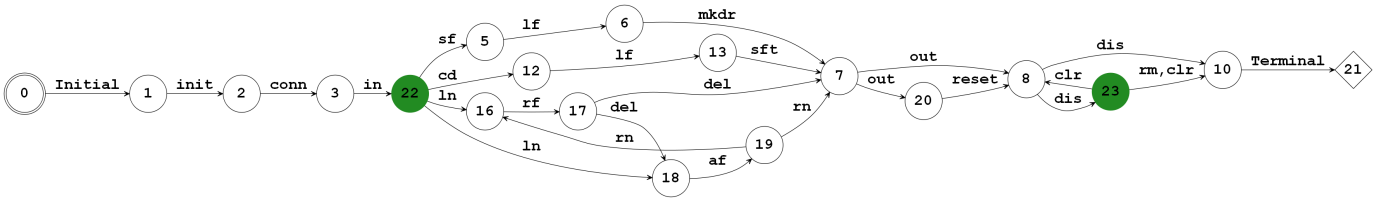

Fig. 5: The result of applying past-equivalence reduction over the mk-Tails model in Fig. 4. The model has 19 states and 27 transitions. Green states correspond to past-equivalent states from Fig. 4.

$I \wedge \forall i < n,\ t_i \in \delta_M \wedge t_{i_e} = t_{i+1_s}$. *Each run defines a word* $w = \langle t_{1_\sigma}, t_{2_\sigma}, \ldots t_{n_\sigma} \rangle$. *The word* w *is in* $\mathcal{L}(M)$ *iff* $t_{n_e} \in F$.

Let M be an FSM over an alphabet Σ. We use $\mathcal{L}(M) \subseteq \Sigma^*$ to denote the set of all words accepted by M.

An FSM is deterministic iff every word has a single corresponding run, and non-deterministic otherwise. We refer to a deterministic, non-deterministic FSM, by the common acronyms DFA and NFA, resp.

B. k-Tails

k-Tails, first introduced in [6], is a classic model inference algorithm. Over the last two decades, k-Tails has been presented in several variants and implemented in many works, e.g., [2], [5], [10], [20]–[22].

k-Tails takes a log and a parameter k as input. It starts by representing the log as an FSM M_{lin} composed of linear sub-FSMs, one per trace, which are joined by adding a single initial state q_{init} transitioning to the start of each trace via a unique α label, and a single terminal state q_{acc} to which all traces transition to at the end via a unique ω label. The language of M_{lin} equals to the language of the log, given that each trace is encapsulated by α and ω events. We refer to this version of the log as the encapsulated version, denoted by L_{en}.

Next, k-Tails iteratively merges states in the M_{lin} FSM: Two states are merged iff they are k-equivalent, i.e., if their future of length k or less, is identical. The algorithm terminates and outputs the resulting FSM when no two remaining states are k-equivalent.

More formally, we define a function $future_k : Q_{M_{lin}} \rightarrow 2^{\Sigma_{\leq k}}$, mapping states in M_{lin} to sets of k-sequences, consecutive sequences of k events or less. The k-equivalence relation induces a partition of the states of the initial FSM M_{lin} into equivalence classes $E = \{e_1, e_2, \ldots, e_m\}$, where each $e_i \in E$ is uniquely defined by its future sequences of length k or less. Two states $q_1, q_2 \in e_i$ iff $future_k(q_1) = future_k(q_2)$.

When lifted from $Q_{M_{lin}}$ to E, the function $future_k$ becomes the injective function $id : E \rightarrow \mathcal{P}(\Sigma_{\leq k})$. For all $q \in e_i$, $future_k(q) = id(e_i)$.

Definition 3 (k-FSM)**.** *k-FSM, the FSM computed by k-Tails for a log L and a positive integer k, is an FSM $M_L = \langle Q, I, F, \Sigma, \delta \rangle$ where: $Q = E$, where E is the set of equivalence classes defined above; Σ is the alphabet of the log L_{en}; $\forall e \in E, a \in \Sigma$: $\delta(e, a) = \bigcup \{e' | \exists q, q' \in M_{lin}$ s.t. $q' \in \delta_{M_{lin}}(q, a) \wedge q \in e \wedge q' \in e'\}$; $I = \{q_{init}\}$ is an artificial initial state; and $F = \{q_{acc}\}$ is an artificial terminal state.*

Among other properties, the correctness of the k-Tails algorithm implies that the k-FSM M may over-approximate but not under-approximate the set of traces in the log L, i.e., every $w \in L$ is accepted by M_L: $L \subseteq \mathcal{L}(M_L)$. Additional useful properties of the k-FSM are that all its states are reachable from the initial state q_{init}, and that the accepting state q_{acc}, is reachable from all states.

Alg. 1 presents pseudo-code for k-Tails implementation. The input of Alg. 1 is a linear representation of an encapsulated log L_{en} and a natural number k, and its output is a k-FSM. The main procedure calls Alg. 1-A. The algorithm initializes an empty dictionary that maps future to futures. Then, it iterates over each state of M_{lin}, computes its future and the future of its consecutive state, and updates the dictionary. Afterwards, the main procedure calls Alg. 1-B, which infers the model. The algorithm starts by initializing an FSM model M. It then iterates over the future dictionary (lines 3-7, Alg. 1-B). The algorithm adds a state every time a new future is encountered, and adds a transition between every pair of states with consecutive futures. The label over the transition is the first event over the sequences that correspond to the equivalence class of the source state. Note that all event sequences in a future equivalence class start with the same event, due to the nature of M_{lin} over which the future equivalence is applied. Thus, the procedure is unambiguous and well defined. Finally, the algorithm unifies all states that are followed by α, ω (the artificial initial and terminal symbols resp.) to a dedicated start and terminal states, and returns the k-FSM model.

Complexity. Since a trace of a total of n events is represented by n unique states in M_{lin} (and the artificial initial and terminal states), constructing and storing M_{lin} requires $O(|L_{en}|_e)$. Iterating over all states of M_{lin} and computing the futures dictionary requires $O(|L_{en}|_e \cdot k)$. Iterating over the futures dictionary requires also $O(|M_\delta|)$, where $|M_\delta|$ is the number of transitions in the k-FSM. Since for any k, $|M_\delta| \leq O(|L_{en}|_e)$, the time complexity of the algorithm is $O(|L_{en}|_e)$. As for the space complexity, storing M_{lin}, the model, and the futures dictionary is bounded by $O(|L_{en}|_e \cdot k)$.

Algorithm 1 kTails

```
1: function ALGORITHM: KTAILS
   input: M: M_lin, Int: k
   output: FSM
2:     Set⟨Set⟨str[]⟩⟩: futures_dict=kFutureMapping(M_lin, k)
3:     return InferModel(futures_dict)
1: function ALGORITHM(A): KFUTUREMAPPING
   input: FSM: M_lin, Int: k
   output: Set⟨str[]⟩ → Set⟨Set⟨str[]⟩⟩: futures_dict
2:     Set⟨str[]⟩ →Set⟨Set⟨str[]⟩⟩: futures_dict = init()
3:     for State: q ∈ Q_{M_lin} do
4:         State: q' = GetNxtState(q, M_lin)
5:         Set⟨str[]⟩: future_q = ComputeKFuture(M_lin, k, q)
6:         Set⟨str[]⟩: future'_q = ComputeKFuture(M_lin, k, q')
7:         futures_dict[future_q].add(future'_q)
8:     return futures_dict
1: function ALGORITHM(B): INFERMODEL
   input: Set⟨str[]⟩ → Set⟨Set⟨str[]⟩⟩: futures_dict
   output: FSM
2:     FSM: M = EmptyFSM()
3:     for Set⟨str[]⟩: ftr_src ∈ futures_dict.keys() do
4:         for Set⟨str[]⟩: ftr_trg ∈ futures_dict[ftr_src] do
5:             AddEqState(M, ftr_src)
6:             AddEqState(M, ftr_trg)
7:             AddTransition(M, ftr_src,ftr_trg)
8:     SetInitialAndTerminalStates(M)
9:     return M
```

C. NFA Reductions

For a given NFA, the problem of finding its minimal language-equivalent NFA is PSPACE-complete [29]. Therefore, to reduce the size of an NFA while preserving its language, past research has focused on heuristics. Ilie and Yu [14] present an algorithm for NFA reduction using invariant equivalences.

An equivalence relation over Q defines a partition over Q, so the terms are used interchangeably.

Formally, a partition over Q is denoted by $\rho = \{B_1, B_2, \ldots, B_n\}$, where B_i is a block of states from Q, $\bigcup_{1 \leq i \leq n} B_i = Q$, and $\forall i, j, B_i \cap B_i = \emptyset$.

One may define a partial order $\subseteq$ between partitions. Let ρ, ρ' denote two partitions of Q. Then, $\rho \subseteq \rho'$ iff $\forall b \in \rho \; \exists b' \in \rho', b \subseteq b'$.

Definition 4. *An equivalence relation ($\equiv$) over Q is right-invariant w.r.t. M iff:*

1) $\equiv \subseteq (Q \setminus F)^2 \cup F^2$ *(terminal and non-terminal states are not equivalent)*
2) $\forall q, q' \in Q, \sigma \in \Sigma, q \equiv q' \rightarrow \delta(q', \sigma) \equiv \delta(q, \sigma)$ *(equivalent states lead to equivalent states on any letter)*

Ilie and Yu [14] show that merging states that are equivalent by any right-invariant relation does not change the language of an NFA, and that there exists a unique largest right-invariant partition over the states of an NFA.

They present a polynomial time algorithm that computes the largest right-invariant equivalence. The algorithm starts from a partition that separates non-terminal and terminal states.

Then, it iteratively searches for none-equivalent states (w.r.t. to Def. 4) and refines the partition until reaching a fixed-point. Finally, equivalent states are merged to obtain a smaller NFA.

In a later work, Ilie et al. [13] show that the largest right-invariant relation $\equiv_R$ is the coarsest stable refinement of the partition $\{F, Q \setminus F\}$ w.r.t. δ. Hence, it can be computed using well-known existing partition refinement algorithms by Kanellakis and Smolka [15].

Importantly in our context, the algorithm can be dually applied for left-invariant equivalences, which are symmetrically defined over the reversed automaton. Both reductions preserve the language of the NFA, while empirically reducing it by 10%-40%. Our past-equivalence reduction tailors the left-invariant reduction for k-Tails.

IV. Contribution: Many Parameters

We now present the first contribution of our work, mk-Tails, a generalization of k-Tails from a single parameter, k, to many parameters, k_1 to k_t.

Based on domain knowledge or on the task at hand, some events may be considered by the engineer as more important or more sensitive than others. For example, when analyzing android apps, sensitive events may be API calls that access sensitive resources [2]. Our generalized version allows the engineer to vary the strength of the abstraction around different subsets of the alphabet, making it less accurate around some and more accurate around others.

A. Defining k-Tails with Many Parameters

The mk-Tails algorithm generalizes the original k-Tails. As input, it takes a log, a set of distinct positive integer parameters $K = \{k_1, \ldots, k_t\}$, and a corresponding partition of the alphabet Σ into disjoint subsets $S = \{S_1, \ldots, S_t\}$. The output of mk-Tails is a k-FSM model that over-approximates the log. Roughly, the algorithm ensures that for every $S_i \in S$, every event $\sigma_j \in S_i$ is associated with equivalence classes with a future length of at least k_i. To formalize, we define the intersection of set of sequences $seqs \in 2^{\Sigma^{\leq k}}$ with a set of events $S \in \mathcal{P}(\Sigma)$ using a boolean function as follows:

$$seqs \cap S = \begin{cases} True & , \exists \sigma. \in S \; \exists seq \in seqs. \; \exists j. \, seq_j = \sigma \\ False & , \text{otherwise} \end{cases}$$

In other words, the set S intersects with $seqs$ iff one of the events in S appears in a sequence in $seqs$.

We now lift the k-Tails equivalence relation by replacing the single k-future function $future_k$ (see Sect. III) with a generalized many-k function.

Definition 5 (*gen-future* function)**.** *Denote the maximal value of k in K by k_{max}. The function gen-future $: Q_{M_{lin}} \times (K, S) \rightarrow 2^{\Sigma_{\leq k_{max}}}$, maps states in M_{lin} to k-sequences of length at most k_{max}, where each state $q \in Q_{M_{lin}}$ is mapped to sequences, $future_{k_i}(q)$, of length k_i, s.t. $S_i \cap future_{k_i}(q) = True$ and $\forall j \neq i \wedge k_i < k_j, S_j \cap future_{k_j}(q) = False$.*

Intuitively, when using the *gen-future* function, equivalence classes that are followed by sequences (see Def. of id in Sect. III) that include an event from S_i, are based on k-futures of at least k_i.

Similar to the original *future* function, *gen-future* induces a partition of the states of M_{lin} into equivalence classes $E = \{e_1, e_2, \ldots, e_m\}$, where each of the equivalence classes in E is uniquely defined by its future sequences. Two states $q_1, q_2 \in e_i$ iff $gen\text{-}future(q_1) = gen\text{-}future(q_2)$.

Definition 6 (mk-FSM)**.** *The mk-FSM is a generalization of the classical k-FSM, which is obtained when replacing the classical k-future function (Sect. III) with the gen-future function.*

Example 2. *Consider the k-FSM in Fig. 4, where K, S are defined as follows: $(k_1, S_1) = (2, \{\texttt{lf}\})$ and $(k_2, S_2) = (1, \Sigma \setminus S_1)$. Consider trace `Tr3` in Fig. 1, and its corresponding branch in M_{lin}, $(q_I^3, q_1^3 \ldots, q_T^3)$. Let us denote the state preceding the fourth event on this branch, `cd`, by q_4^3. Clearly, $future_1(q_4^3) = \{[\texttt{cd}]\}$, and $future_2(q_4^3) = \{[\texttt{cd}], [\texttt{cd}, \texttt{lf}]\}$. To compute gen-future$(q_4^3)$, we compute $S_1 \cap future_{k_1}(q_4^3)$, and $S_2 \cap future_{k_2}(q_4^3)$, and take the maximal k among the pairs (k_i, S_i) that return true. Since both return true, we get that gen-future$(q_4^3) = \{[\texttt{cd}], [\texttt{cd}, \texttt{lf}]\}$.*

Remark 1. *The mk-FSM defined by mk-Tails, M, has similar properties to the classical k-Tails model k-FSM. Specifically, again, for a log L, every $w \in L$ is accepted by M, i.e., $L \subseteq \mathcal{L}(M_L)$; all the states of M are reachable from the initial state q_{init}; and the accepting state, q_{acc}, is reachable from all states. In particular, when $S = \{\Sigma\}$ and $K = \{k\}$, the output of mk-Tails is the same as the output of the original k-Tails. In this sense, mk-Tails is a conservative extension of k-Tails.*

B. Computing the mk-FSM

To compute the generalized k-FSM, mk-FSM, we use Alg. 1, but replace the calls to `ComputeKFuture` with the function `ComputeGenFutures`, shown in pseudo-code in Alg. 2. Similar to `ComputeKFuture`, as input, `ComputeGenFutures` receives a model and a state q. However, instead of a single k, the function receives an array of pairs. Each pair $\langle k_i, S_i \rangle$ consists of an integer $k_i \in \{k_1, \ldots, k_t\}$ and a corresponding $S_i \subseteq \Sigma$. The different S_i are disjoint and their union equals Σ, formally $\bigcup_{i=1}^{t} S_i = \Sigma$. The function assumes a descending order of the pairs on the first component. This ensures that for each state q, Alg. 2 avoids redundant computations of futures with length less than the actual k required for q.

Algorithm 2 ComputeGenFutures

```
1: function ComputeGenFutures(M, k_sets_arr, q)
   input: M_lin: M, Array⟨K, S⟩: k_sets_arr (sorted desc. by k), State: q
   output: future_q: Set⟨str[]⟩ # future of length k_i ∈ K
2:     for k_i, S_i ∈ k_sets_arr do
3:         future(k_i)_q = ComputeKFuture(M, k_i, q)
4:         if S_i ∩ future(k_i)_q then return future(k_i)_q
```

Complexity. Denote the size of the pairs array by t. To sort it, the algorithm requires $O(t\log(t))$. This sorting is performed once. Then, the only difference from Alg. 1 are the calls to Alg. 2. We denote by k_{max} the maximal k in K. The algorithm makes at most t loop iterations, in which it makes at most k_{max} steps (computes k-future by the `ComputeKFuture` func.) and performs an intersection check. The `intersects`(S_i,$future(k_i)_q$) function requires $k_i \cdot |S_i|$ steps. Therefore, the extended algorithm adds a factor that is bounded by $t \cdot max(K) \cdot |S_{max}|$ per M_{lin} state, where $max(K)$ and S_{max} denote the largest k, and the largest subset of events resp. Therefore, the overall complexity is $O(t \cdot max(K) \cdot |S_{max}| \cdot |L_{en}|_e + t\log(t))$. Since Alg. 1 complexity is $O(|L_{en}|_e \cdot k)$ (see Sect. III), we get that the increase in the complexity is only the multiplication by a constant $\dot{k} = t \cdot max(K) \cdot |S_{max}|$, i.e., $O(|L_{en}|_e \cdot \dot{k})$.

V. Contribution: k-FSM Reduction

We now present the second contribution of our work, an effective and efficient size reduction technique that preserves the language of the k-FSM model.

A. Right and Left-Invariant Equivalence for k-FSM

To reduce the size of the k-FSM, one may consider using the right-invariant equivalence (see Def. 4) and its symmetrical left-invariant equivalence. These may be useful, since merging states by these relations is language preserving (Ilie and Yu [14]). Below we will show that in the context of k-FSM, the right-equivalence reduction has no effect. Then, in contrast, we will show that in the context of k-FSM, the left-equivalence reduction is effective and can be efficiently computed.

Theorem 1. *Consider a k-FSM M, and a pair of states $e, e' \in Q_M$. Let $\equiv_R$ denote a right-invariant equivalence relation over Q. Then, $e \not\equiv_R e'$.*

Proof. Assume towards a contradiction two different states e and e' s.t. $e \equiv_R e'$. Condition 2 of Def. 4 implies that any transition from q can be matched by an equivalent transition from q'. Therefore, by inductively applying $\equiv_R$, we get that for any sequence $seq_k = \langle e, e_1, \ldots, e_k\rangle$ that belongs to $id(e)$, there exists a sequence $seq'_k = \langle e', e'_1, \ldots, e'_k\rangle$ s.t. $e_i \equiv_R e'_i$. Since R is an equivalence relation, the same argument holds from any $seq'_k \in id(e')$. Hence, e and e' have similar futures, i.e, $id(e) = id(e') \rightarrow e = e'$, which contradicts the assumptiona that e and e' are two different states. □

Corollary 2. *As an immediate corollary, any k-FSM model is minimal w.r.t. any right-invariant relation (i.e., no two states can be merged).*

The implication of Corollary 2 is that using any right-invariant relation for reducing a k-FSM M is ineffective.

In contrast to the right-invariant relations, merging states by left-invariant relations is a very effective method for reducing the size of a k-FSM. As an example, consider a log with the following three traces $t_1 = \langle a, b, c\rangle$, $t_2 = \langle a, b, d\rangle$, $t_3 = \langle a, b, e\rangle$. Fig. 6 (top) shows the output of running k-Tails with

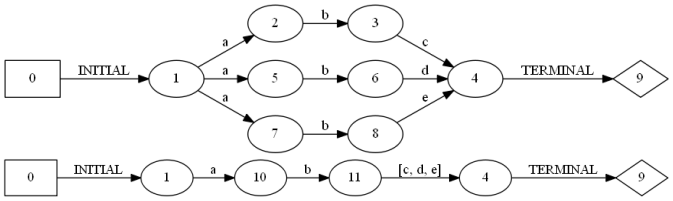

Fig. 6: A k-FSM (top) obtained by running k-Tails over $L = \{\langle a, b, c\rangle, \langle a, b, d\rangle, \langle a, b, e\rangle\}$ with $k = 2$, and its left-invariant reduction (bottom)

$k = 2$ over the log. As can be seen, states $\{2, 5, 7\}$ are left-equivalent, as they all have a single incoming transition from state 1 with the label *'a'*. Therefore, we can merge them without changing the language of the model. Note that k-Tails splits those states due to the future divergence in states $\{3, 6, 8\}$. Such redundant splits are common in k-FSMs. After applying left-equivalence once, the succeeding states $\{3, 6, 8\}$ become left-equivalent and can be merged, which results in the model in Fig. 6 (bottom).

In its special application to k-Tails, we call the left-invariant equivalence *past-equivalence*.

B. Computing Past-Equivalence Reduction of a k-FSM

We now present an efficient algorithm for computing past-equivalence reduction of k-FSM. Most importantly, our algorithm exploits the structure of the k-FSM. In contrast to the algorithm presented by Ilie and Yu [13], [14], which starts from a coarse partition and employs refinement operations until convergence, our algorithm starts from a refined partition and employs an iterative coarsening operation until convergence. While our procedure is general, it converges on any k-FSM after at most k+1 iterations, yielding an improved complexity over the state-of-the-art.

Alg. 3 presents the pseudo-code for past-equivalence reduction implementation. The algorithms input is the *futures_dict* constructed by Alg. 1-A, which maps future to futures. Recall, that each future corresponds to a state in the k-FSM, and is represented by a set of strings of length k_{max} or less.

The algorithm uses the *futures_dict* to construct δ (line 2) from which it computes the transitions that are incoming each state using the `ComputeIncomingTransitions` function. Then, the algorithm initializes an initial partition of the states, where every state is mapped to a single state block, with a unique block id (lines 5-7).

The algorithm performs a series of block merging iterations (Alg. 3, lines 8-27), until reaching a fixed-point (no more blocks can be merged). At the beginning of each iteration, the algorithm computes the *pst_eqv_states* dictionary, which maps the sets of incoming transitions (at the block level) to states that succeed them (lines 9-16). To this end, the algorithm iterates over the *incm_dict* dictionary, and computes the set of incoming transitions per state using the *state2block* dictionary. Then, it calls `AddPastSet`, which adds the state q to the past-equivalence class of *block_in_trns* (line 16). Importantly, after iterating all states, this dictionary holds past-invariant equivalence classes w.r.t. the current partition. Then,

Algorithm 3 ComputePastEquivalence

```
1:  function ALGORITHM: COMPUTEPASTEQUIVALENCE
    input: Set⟨str[]⟩ →Set⟨Set⟨str[]⟩⟩: futures_dict
    output: State→Int: state2block
2:      Set⟨Transition⟩: δ = GetKFSMTransitions(futures_dict)
3:      State→Set⟨Transition⟩: incm_dict = /
4:                              ComputeIncomingTransitions(δ)
5:      Int: b_id = 0
6:      State→Int: state2block = {}
7:      for q ∈ incm_dict.keys() do state2block[q] = b_id++
8:      while True do
9:          # compute past-equivalence by current partition
10:         Set⟨Transition⟩→Set⟨State⟩: pst_eqv_states = {}
11:         for State: q ∈ incm_dict.keys() do
12:             Set⟨(Int, σ)⟩: block_in_trns = ()
13:             for Transition: t ∈ incm_dict[q] do
14:                 Int: inc_block_id = state2block[t_src]
15:                 block_in_trns.add((inc_block_id, t_σ))
16:             AddPastSet(pst_eqv_states, block_in_trns, q)
17:         # check if new past-equivalence found
18:         Bool: fixed_point = True
19:         for Set⟨Transition⟩: pst_eqv ∈ pst_eqv_states do
20:             Set⟨State⟩: states = pst_eqv_states[pst_eqv]
21:             blocks = ()
22:             for q ∈ states do blocks.add(state2block[q])
23:             if |blocks| > 1 then
24:                 fixed_point = False
25:                 # coarsen partition
26:                 b_id += 1
27:                 for q ∈ states do state2block[q] = b_id
28:         if fixed_point then return state2block
```

the algorithm iterates over each past-equivalence class and checks if it includes states that belong to more than a single block according to the current partition (lines 19-27). If so, it merges such states by mapping them to a new block id, and updates the *state2block* dictionary (lines 26-27). Finally, the algorithm reaches a fixed-point when no new block is added, and returns the last partition (*state2block*).

The correctness of our algorithm is based on the fact that it finds the unique coarsest past-invariant equivalence. Its complexity of $O(|\delta| \cdot k)$ is based on the fact that it reaches the fixed-point after at most k+1 iterations, and in each iteration (lines 8-28) it iterates over $|\delta|$ twice.

Example 3. *Consider our example k-FSM in Fig. 6, which was generated by running k-Tails with k=2. We focus on the states that are merged. Initially, the algorithm computes the incoming transition dictionary (incm_dict), and maps each state to a singleton block (line 2-3). Let us denote the initial block of each state by the state id. At the start of the first iteration, the algorithm computes the incoming transitions at the block level (lines 9-16) and updates the past-equivalence dictionary (pst_eqv_states). States $\{2, 5, 7\}$ all have a single incoming transition from block b_1 with an 'a' label. Therefore, they are added to a single equivalence class (line 16).*

Then, the algorithm loops over the past-equivalence sets in pst_eqv_states, and maps states that are grouped together and are associated with different blocks, to a new block id (lines 19-27). States $\{2, 5, 7\}$, which do not share a block, are mapped to a new block, b_{10} (line 27). Since a new block was added, the algorithm makes another iteration.

On the second iteration, the algorithm recomputes the past-equivalence dictionary. This time, states $\{3, 6, 8\}$ all have a single incoming transition from b_{10} with a 'b' label. Therefore, they are added to a single equivalence class, and are later merged to a new block b_{11}.

Finally, the algorithm makes one last iteration. Since no two other states are past-equivalent, the algorithm reaches a fixed-point after 3 iterations, and returns the following mapping: $\{0\} \to b_0, \{1\} \to b_1, \{2, 5, 7\} \to b_{10}, \{3, 6, 8\} \to b_{11}, \{4\} \to b_4, \{5\} \to b_5$.

We list three theorems by which we state the correctness and complexity of Alg. 3. The theorems are stated w.r.t. a k-FSM, but trivially hold for a mk-FSM. Formal proofs are in supporting materials [1].

Theorem 3. *Algorithm 3 terminates after at most k+1 iterations on any k-FSM, with a maximal future of k.*

Theorem 4. *Let ρ_i, ρ_{i+1} denote the partitions obtained by the iteration$_i$, iteration$_{i+1}$ of Alg. 3 resp. Then, $\rho_i \subseteq \rho_{i+1}$ (see Sect. III), and ρ_{i+1} unifies any blocks in ρ_i that are past-equivalent.*

Theorem 5. *Algorithm 3 terminates on the coarsest past-equivalence partition ρ_{max}, where ρ_{max} is the partition such that $\forall \rho$, if ρ is a past-equivalence partition, then $\rho \subseteq \rho_{max}$.*

VI. Implementation, Validation, and Evaluation

A. Implementation and Validation

We implemented k-Tails, mk-Tails, and the past-equivalence reduction in Java. The end-to-end implementation allows the engineer to set up a mapping between subsets of the alphabet and their corresponding ks, and to apply k-FSM past-equivalence reduction. It computes and visually presents a k-FSM similar to the one in Fig. 3. We made it available as a prototype web application for review and experiments. We encourage the interested reader to check it out in supporting materials [1].

Further, to test the scalability of past-equivalence reduction (Alg. 3), we compared it against Kanellakis and Smolka [15] (see Sect. III-C). We followed the description of the algorithm and implemented it in Java.

We validated our implementation as follows. First, we run small examples like the ones used in Sect. II, where we were able to manually inspect and validate the correctness of the output. Second, we added tests for the following assertions: (1) each trace from the log is accepted by the model, (2) the past-equivalence reduction does not change the language of the k-FSM, (3) the past-equivalence reduction yields identical

model to that of Kanellakis and Smolka [15]. We used Brics [7], a well-known automaton Java library, to validate assertions 1 and 2, and compared the number of states and transitions in the reduced models to validate assertion 3. The validations increased our confidence in the correctness of our ideas and implementation.

B. Research Questions

We consider the following research questions:

RQ1 Can we *efficiently compute* mk-Tails?

RQ2 Can we *reduce model size* and *increase conditional accuracy* around sensitive events using mk-Tails?

RQ3 Can we *efficiently compute* past-equivalence reduction over mk-Tails?

RQ4 Can past-equivalence reduction over mk-Tails *effectively reduce* the size of the model?

C. Logs, Setup, and Measures

`SET1`. We used 10 finite-state machine models in our evaluation, all taken from the literature [17], [20], [25]–[27]. The models vary in size and complexity, i.e., the alphabet size ranges from 7 to 42, the number of states ranges from 6 to 22, and the number of transitions ranges from 15 to 209. From these 10 models we generated logs using a publicly available trace generator from Lo et al. [20], configured to provide state coverage of four visits per state and a minimum of 1000 traces. These yielded logs of roughly 1000 traces each. The complete list of models and their statistics, and the generated logs, are available in supporting materials [1].

`SET2`. In addition, we used six real-world logs we have obtained from a large telecommunication company. These logs have an alphabet size that ranges from 21 to 46, number of traces from 42 to 204, number of events from 1169 to 9252, and average trace length from 38.43 to 99.98. The logs (anonymized) are available in supporting materials [1].

To evaluate mk-Tails, we selected a subset of events from the log. We defined a random set of events to be "sensitive" events, and assigned them with a higher k than the rest of the events.

To measure the ability of mk-Tails to reduce noise around sensitive events, we follow a similar procedure to the one suggested by Lo and Khoo [17].

First, we define the notion of precision. Let us consider a log L and the mk-Tails model M inferred from it. Let us consider a sample of the traces from M, and denote it by L'. We define the precision of the model as the fraction of traces from L' that appear in the log L, i.e., P=$|L' \cap L|/|L'|$.

Since we focus on the sensitive events, we extend the notion of precision to conditional precision, which only accounts for traces containing at least one sensitive event.

More formally, let S be a set of sensitive events. Further, let us denote by L'_S, the set of traces from L' that include events from S, i.e., $L'_S = \{t | t \in L' \wedge \exists e \in S \text{ s.t. } e \text{ appears in } t\}$. Then, the conditional precision w.r.t. S is defined as follows: CP=$|L'_S \cap L_S|/|L'_S|$.

Remark 2. *Since k-Tails models are an over-approximation of the log, their recall w.r.t. the log equals 1. Hence, we did not compute it.*

Remark 3. *We chose to compare precision w.r.t. the log and not w.r.t. the model, as was done by Lo and Khoo [17]. We do so, since our underlying assumption that larger ks yield more accurate models holds w.r.t. the log, and not w.r.t. the model. We do not make any claims about the generalization capabilities of k-Tails to learn new behaviors from the observed ones, but merely focus on its ability to compactly capture behaviors that appear in the log, without introducing much noise.*

To measure the ability of our algorithm to reduce the model size, we define the model size reduction. Let M_1 and M_2 denote two models. We denote by $|M|$ the total number of states and transitions in M. Then, the size reduction of M_1 w.r.t. M_2 is defined as 1-$|M_1|/|M_2|$.

Finally, we measure the running times. In measuring the running time we include the time of parsing the logs and computing the models. In RQ1 and RQ2, by mk-Tails, we refer to Alg. 1 combined with Alg. 2. In RQ3 and RQ4, we refer to Alg. 3. We executed all experiments on an ordinary laptop computer, Intel i7 CPU 2.6GHz, 16GB RAM with Windows 10 64-bit OS, Java 1.8.0_161 64-bit. We executed all runs at least 10 times, to average out measurement noise from the Java execution. We report average and median running times.

D. Experiments and Results

RQ1 To answer RQ1, we conducted the following experiment. We run k-Tails and mk-Tails on the logs of `SET1` and `SET2`. For k-Tails we used $k = 2$. For mk-Tails we randomly selected 10% of the log alphabet as sensitive events. For mk-Tails, for the none sensitive events we used $k = 2$, and for the sensitive events we used $k' \in \{3, 4, 5\}$ (in three different runs). We measured the running times. In this experiment, we did not include the log reading time, which is common to both algorithms.

Fig. 7 shows the average running times of k-Tails and mk-Tails for each of the logs (`SET1` followed by `SET2`), in milliseconds. For each log, the blue and orange columns depict k-Tails and mk-Tails (Alg. 1 and Alg. 2) average running times resp. In all, the running time for mk-Tails was higher than the one for k-Tails. Overall, mk-Tails adds an overhead that ranges from 69.55% to 465.58%, with an average increase of 259%.

Finally, in absolute terms, Fig. 7 shows that the total average running times of mk-Tails in all logs is below 200 milliseconds. This demonstrates the applicability of mk-Tails to different logs of realistic sizes.

To answer [RQ1], computing mk-Tails does not come for free, however, the overhead above k-Tails is limited. Running mk-Tails with 10% of its alphabet defined as sensitive, for a variety of different logs with thousands of traces and tens of thousands of events, did not take more than a second.

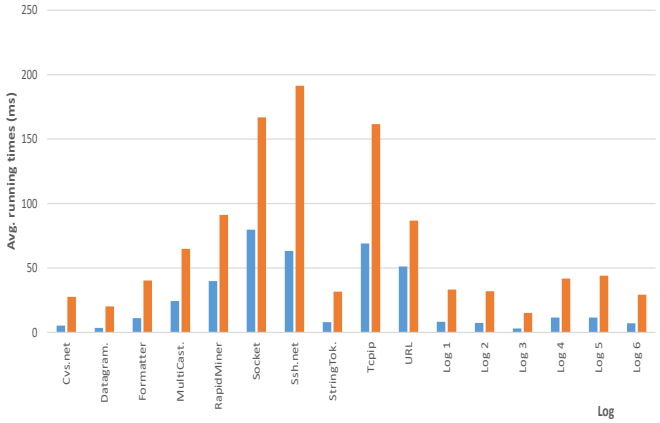

Fig. 7: Average running times (ms) of k-Tails (blue), mk-Tails (orange) for $k = 2$, $k' \in \{3, 4, 5\}$, and 10% of alphabet as sensitive events.

TABLE I: mk-Tails conditional precision and size reduction

Log	k=1	k'=3		k'=4		k'=5	
	CP	CP	Size Red.	CP	Size Red.	CP	Size Red.
RapidMiner	0.32%	39.40%	41.62%	39.44%	64.33%	40.22%	73.14%
CVS	0.98%	96.52%	32.07%	96.52%	50.45%	96.52%	62.24%
DatagramSocket	0.00%	86.02%	33.64%	89.16%	37.78%	90.38%	37.59%
MultiCastSocket	0.03%	76.24%	63.06%	84.34%	66.15%	85.26%	64.50%
Socket	0.00%	99.08%	59.50%	99.38%	66.38%	99.48%	65.64%
URL	0.00%	86.38%	64.23%	88.00%	64.55%	88.10%	60.74%
Formatter	0.16%	39.24%	45.57%	67.44%	47.13%	74.40%	43.40%
StringTokenizer	0.28%	36.40%	36.18%	70.14%	35.40%	77.28%	29.93%
SSH	0.10%	53.48%	15.93%	61.34%	14.91%	64.30%	11.48%
TCP/IP	0.08%	46.02%	52.85%	61.14%	60.03%	78.26%	54.99%
L1	0.00%	81.58%	50.03%	88.34%	64.00%	91.54%	69.59%
L2	0.00%	88.64%	47.82%	95.88%	59.35%	98.26%	61.79%
L3	0.00%	94.90%	31.55%	98.90%	46.43%	99.68%	55.36%
L4	0.00%	97.12%	51.23%	98.36%	71.92%	99.20%	80.21%
L5	0.01%	88.50%	49.29%	95.68%	61.60%	98.72%	64.49%
L6	0.03%	96.84%	51.25%	98.80%	64.49%	99.00%	67.85%

RQ2 To answer RQ2, we compared the models produced with mk-Tails against models produced with a single k in terms of size and conditional precision (CP), by conducting the following experiment. We run k-Tails, setting $k = 1$ as a baseline. Then, we randomly selected a single event from the alphabet of each log as sensitive event. To measure the effectiveness of mk-Tails in improving the conditional precision, we first computed the precision of the baseline model (i.e., $k = 1$). We run a random trace generator (Lo et al. [20]) over it, and produced 500 traces that included the sensitive event. Then, we measured its conditional precision (Table I, second column from the left) w.r.t. the original log.

To measure the ability of mk-Tails to filter noise around sensitive events, we measured its conditional precision w.r.t. the log produced by the baseline model ($k = 1$) (Table I, columns 3, 5, 7 from the left). That is, we only accounted for traces that were accepted by both the small k-Tails model and the more refined mk-Tails model. To minimize the potential bias of the random choice of the sensitive event, we repeated the selection of the sensitive event 10 times per log.

As can be seen, the conditional precision varies across models and is much greater in comparison to $k = 1$. Further, as expected, it increases with k': we measured an average CP of 75.39%, 83.30%, 86.28% for $k' \in \{3, 4, 5\}$ resp.

Table I also shows the size reduction in comparison to running k-Tails with an increased k'. That is, we compared the size of mk-Tails models with (k, k') to that of k-Tails models with k'. For example, the value 41.62% in the row for model `RapidMiner`, column $k' = 3$, means that 1-$|M_1|/|M_2| = 0.4162$, where M_1 is the model inferred using mk-Tails with $k = 1$ and $k' = 3$, and M_2 is the model inferred using k-Tails with $k = 3$. As can be seen, mk-Tails is able to dramatically reduce the size of the model across different logs. Further, the reduction gains increase with k', with a median of 47.45%, 59.34%, 60.73% for $k' \in \{3, 4, 5\}$ resp.

To answer [RQ2], we have evidence that mk-Tails can yield significant reduction in the model size while increasing the conditional precision.

TABLE II: Median running times, in milliseconds, of Alg. 3 and Kanellakis and Smolka [15], when running with $k \in \{2, 4, 6, 8, 10\}$ over log sets `SET1` and `SET2`.

k	2	4	6	8	10
Alg. 3	17.0	40.0	62.0	84.0	82.0
Kanellakis and Smolka [15]	93.5	1705.5	13196.5	15784.0	18980.5

RQ3 To answer RQ3, we compared the running time of the general NFA reduction algorithm (Kanellakis and Smolka [15]) with the past-equivalence reduction (Alg. 3) for $k \in \{2, 4, 6, 8, 10\}$ over log sets `SET1` and `SET2`.

Table II reports the median absolute running times of applying past-equivalence reduction with Alg. 3 and Kanellakis and Smolka [15], in milliseconds, as function of k across the logs. As can be seen, Alg. 3 requires an order of magnitude less time than the competing algorithm.

Figure 8 shows a boxplot of the running time as function of k across different logs, where boxes labeled by `PM_k` and `SM_k` correspond to the application of Alg. 3 and Kanellakis and Smolka [15] over k-FSMs resp. Note that the boxplot does not include outliers due to the long running times of the `Socket` logs when running with Kanellakis and Smolka [15]. The boxplot shows that the running times of Kanellakis and Smolka [15] are higher than those of Alg. 3, for all logs and all values of k.

To explain the large difference in running times, we investigated the number of iterations performed by each of the algorithms. Since both algorithms move between partitions via iterations that require $O(|\delta|)$ until reaching a fixed-point, this factor is key in analyzing the complexity of the algorithms.

By Theorem 3, our algorithm converges in at most k+1 iterations. Table III shows the average number of iterations until Kanellakis and Smolka [15] reaches a fixed-point for different values of k across the logs. As can be seen, the number of iterations in Kanellakis and Smolka [15] rapidly increases with k, in particular when moving between small values of k. This explains the superior running times of Alg. 3.

To answer [RQ3], we have evidence that Alg. 3 scales well, and improves over the state-of-the-art in terms of computation time.

TABLE III: Average number of iterations made by Kanellakis and Smolka [15], when running with $k \in \{2, 4, 6, 8, 10\}$ over log sets SET1 and SET2.

k	2	4	6	8	10
Iterations	2.81	132.31	198.87	225.81	234.50

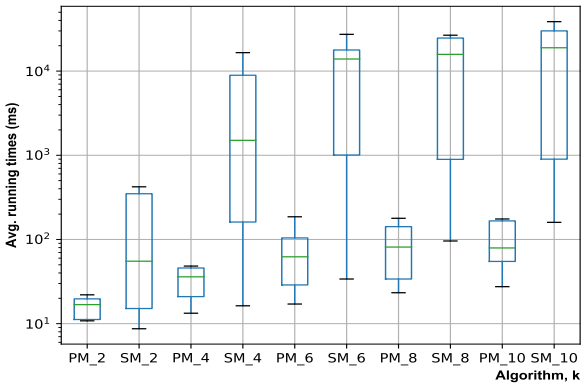

Fig. 8: Average running times (ms) (log scale) of Alg. 3 (PM) and Kanellakis and Smolka [15] (SM) as function of k, when run over SET1 and SET2.

RQ4 To answer RQ4, we report the model size before and after applying past-equivalence reduction. We run the reduction Alg. 3 over models produced with $k \in \{2, 4, 6, 8, 10\}$, over log sets SET1 and SET2.

Table IV shows the median size of the k-Tails models, the median size of the reduced k-Tails models, and the median size reduction per model, for different values of k.

As can be seen, the algorithm effectively reduces the size of the models across all k values. The median size reduction reduces with k, but is above 15% across all values of k. Further, the measured size reductions are consistent with the ones reported by Ilie and Yu [14] and are in the range $10\% - 40\%$.

To answer [RQ4], we have evidence that Alg. 3 effectively reduces the size of k-Tails models. The algorithm achieves significant size reduction across different logs and for different values of k.

Threats to validity. We consider the following threats to the validity of our results. First, our implementation may have bugs. To mitigate this, we validated our implementation extensively, see Sect. VI-A. Second, the logs used may not be representative of real-world logs. To address this, we used a sample of real-world logs from a large telecommunication company and synthetic logs we generated from real-world models borrowed from prior works, with high variability in number of events, trace length, etc., see Sect. VI-C.

VII. Discussion and Related Work

We discuss alternatives to our solutions as well as limitations and implications of our work. We then continue to discuss related work.

TABLE IV: Size reduction over log sets SET1 and SET2.

k	2	4	6	8	10
k-Tails model size (median)	1411.0	4720.5	6294.0	6868.0	6920.0
Reduced k-Tails model size (median)	604.0	2863.0	5424.0	5960.0	6065.0
Size reduction per model (median)	36.85%	22.37%	28.40%	21.49%	16.82%

A. Discussion

Alternatives to the many parameters of mk-Tails. One may suggest other methods to focus on events of interest. First, for example, filtering on the log (e.g., ignoring infrequent events or traces, or ignoring events that are considered uninteresting). Second, slicing on the finite-state machine model [3].

We view these methods as complementary to mk-Tails. mk-Tails allows fine-grained control over the accuracy around the selected more sensitive events while not giving up the context that is often lost when using filtering or slicing techniques.

Alternative reduction algorithm. Ilie et al. [13] note that one may use the well-known Paige-Tarjan (PT) algorithm [24] to apply the past-equivalence reduction. This algorithm has time complexity of $O(|\delta|log(|Q|))$, which is better than the $O(|\delta||Q|)$ of Kanellakis and Smolka [15]. Both algorithms, however, use the same approach of starting from a coarse partition and apply a series of split iterations, each with a cost of $O(|\delta|)$.

In contrast, our algorithm uses an opposite approach. It starts from a refined partition and, importantly, applies only at most k+1 state merging iterations (each with the same cost of $O(|\delta|)$). The large number of iterations made by Kanellakis and Smolka [15] (see Table III) indicates that reducing the number of iterations to a constant rather than to a log of the number of states is expected to be competitive. We leave a direct comparison with PT to future work.

Is precision monotonic with k? One may expect that increasing the value of k would increase the precision of the inferred model relative to the log. However, this is not necessarily the case. Nevertheless, for every log there is a large enough k such that precision equals 1. This applies to k-Tails as well. As for conditional precision, when one only increases k', a value of 1 may never be obtained, due to possible imprecision entailed by other, non-sensitive, events.

Under which conditions does mk-Tails provide size reduction? mk-Tails is a heuristics. Specifically, assigning a higher value of k to a small subset of the events rather than to all of them, will never increase the model size but does not guarantee to reduce it. For example, when the sensitive events for which we assign the higher k immediately succeed all non-sensitive events, in one or more of the traces. On the other hand, when the sensitive events appear in relative isolation, as part of a small number of unique k-sequences, we will obtain a significant reducution in the model size. In practice, as our experiments demonstrate, mk-Tails is able to dramatically reduce the size of the model across different logs, see RQ2 in Sect. VI.

Implications to anyone who uses k-Tails. mk-Tails allows engineers to control the detailedness of the model in different

areas based on domain knowledge or the specific task at hand. Instead of a single level of abstraction, increasing k improves model accuracy around selected events, while reducing the model size in comparison to a single global k. Although mk-Tails allows much flexibility in choosing different ks to different subsets of the alphabet, in practice, choosing these ks may not be easy. We believe that in practice one would be satisfied with defining only a simple binary partition between "interesting" and "less interesting" events, and assigning a higher value of k to the former and a lower value of k to the latter.

B. Related Work

We discuss model inference works that deal with the notion of model size and accuracy.

Some authors have used precision, recall, and F-measure to quantitatively evaluate the quality of the inferred models [11], [16]–[19], [27], [30]. Lo and Khoo [17], [18] have also used co-emission and PS. In all of these, the inferred models are compared against the ground-truth models from which the logs were produced. The comparison is done by sampling traces from both models and performing acceptance testing. Thus, all these require ground-truth models, which are available in experiments but not available in real-world setup.

In our evaluation of mk-Tails, we followed a different approach to evaluate the accuracy of the generated models. We compared the inferred model against the log, not against the ground-truth model from which the log was generated (see Rem. 3). We demonstrated how increasing the value of k increases accuracy but comes with the price of larger models. We have also defined and used conditional precision, specifically in order to show that mk-Tails is able to increase conditional precision around events of interest while paying a rather low cost in additional model size.

Many works have used, implemented, or extended k-Tails, e.g., [2], [4], [5], [8]–[10], [12], [16]–[23], [27], [28], [31]. We cite them here as evidence for the popularity of k-Tails in the literature, which motivated our choice of this algorithm as a baseline for our work on size vs. accuracy. To our knowledge, none of these works has considered a generalization from one to many parameters and thus all are limited to the single level of abstraction defined by the choice of k. In addition, none has applied post-processing size reductions. Our size reduction is different than the general NFA reduction as its better theoretical complexity and better empirical performance relies on the specific context of the k-Tails algorithm.

VIII. Conclusion and Future Research

To deal with the tradeoff between size and accuracy in model inference, we presented mk-Tails, a generalization of k-Tails from single to many parameters, which enables fine-grained control over the abstraction on different subsets of the event alphabet. We have further presented an efficient algorithm based on past-equivalence, to reduce the size of the resulting model without affecting its accuracy. We implemented our ideas in a prototype web-based application, which we have made available for experiments.

Our evaluation over logs generated from models from the literature and additional logs provided to us by our industrial partner, shows that mk-Tails can be computed efficiently, with only little overhead above the classical k-Tails. Moreover, it shows that the use of mk-Tails can dramatically reduce model size, while maintaining high conditional accuracy for events of interest. Finally, it shows that our past-equivalence reduction, when applied to the models generated by mk-Tails, is efficient and effective in reducing model size.

The important implications for any work that uses k-Tails is to consider finer control over the abstraction by assigning different values of k to different subsets of the alphabet, based on domain knowledge.

We consider the following future research directions. First, to improve the practicality and applicability of our work, it may be interesting to look at (semi-)automated means to translate domain knowledge or data on the task at hand into the selection of the subsets of events to consider "sensitive".

Second, once the subsets of events are given, how should the values of the ks be selected? One may suggest to automate the choice of different ks based on target (conditional) precision given by the engineer. Note that this may indeed be possible, since in our framework, computing precision does not require a ground-truth model.

Third, one could extend our own statistical approach [8] from k-Tails to mk-Tails. Following this work, it may be useful to strengthen accuracy computations with statistical guarantees. Intuitively, when sampling from a large log, we may want to stop sampling when we have enough confidence that the estimated accuracy we have obtained is close enough to the accuracy of the model that one would have inferred from the complete log.

Fourth, note that mk-Tails, like the classic k-Tails, deals with the tradeoff between size and accuracy in a way that abstracts away the frequencies of the different traces or k-sequences in the log. In some domains and for some applications, however, these frequencies are important and should be represented in the inferred model. There, it would be necessary to extend mk-Tails and the notion of accuracy to account for frequencies.

Finally, we consider an interactive application inspired by mk-Tails, where the engineer can dynamically increase and decrease the local value of k on selected states or events. This will result in a dynamic details-on-demand approach to model inference. We leave all these for future work.

Acknowledgments

We thank Or Pistiner for helpful discussions. We thank the anonymous reviewers for their helpful comments. This work is partly supported by the Len Blavatnik and the Blavatnik Family Foundation, and by the Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University.

REFERENCES

[1] Supporting materials website. http://smlab.cs.tau.ac.il/xlog/#ASE19a.

[2] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz. Using finite-state models for log differencing. In *ESEC/SIGSOFT FSE*, pages 49–59, 2018.

[3] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt. State-based model slicing: A survey. *ACM Comput. Surv.*, 45(4):53:1–53:36, Aug. 2013.

[4] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Software Eng.*, 41(4):408–428, 2015.

[5] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *SIGSOFT FSE*, pages 267–277, 2011.

[6] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[7] Brics. https://www.brics.dk/automaton/.

[8] N. Busany and S. Maoz. Behavioral log analysis with statistical guarantees. In *ICSE*, pages 877–887. ACM, 2016.

[9] H. Cohen and S. Maoz. Have we seen enough traces? In *ASE*, pages 93–103. IEEE, 2015.

[10] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.

[11] S. S. Emam and J. Miller. Inferring extended probabilistic finite-state automaton models from software executions. *ACM Trans. Softw. Eng. Methodol.*, 27(1):4:1–4:39, 2018.

[12] M. Goldstein, D. Raz, and I. Segall. Experience report: Log-based behavioral differencing. In *ISSRE*, pages 282–293, 2017.

[13] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. In *Karhumakai J., Maurer H., Paun G., Rozenberg G. (eds) Theory Is Forever. Lecture Notes in Computer Science, vol 3113. Springer, Berlin, Heidelberg*, pages 112–126. Springer, Berlin, Heidelberg, 2004.

[14] L. Ilie and S. Yu. Reducing NFAs by invariant equivalences. *Theoretical Computer Science*, 306(1):373 – 390, 2003.

[15] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

[16] T. B. Le, X. D. Le, D. Lo, and I. Beschastnikh. Synergizing specification miners through model fissions and fusions. In *ASE*, pages 115–125. IEEE, 2015.

[17] D. Lo and S.-C. Khoo. Quark: Empirical assessment of automaton-based specification miners. In *WCRE*, pages 51–60. IEEE Computer Society, 2006.

[18] D. Lo and S.-C. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, pages 265–275, 2006.

[19] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *ESEC/SIGSOFT FSE*, pages 345–354. ACM, 2009.

[20] D. Lo, L. Mariani, and M. Santoro. Learning extended FSA from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063–2076, 2012.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[22] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Trans. Software Eng.*, 37(4):486–508, 2011.

[23] L. Mariani and M. Pezzè. Dynamic detection of COTS component incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[24] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, Dec. 1987.

[25] E. Poll and A. Schubert. Verifying an implementation of SSH. In *WITS*, volume 7, pages 164–177, 2007.

[26] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[27] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *ICSM*, pages 1–10, 2010.

[28] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, pages 221–230, 2001.

[29] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer-Verlag, Berlin, Heidelberg, 1997.

[30] N. Walkinshaw and K. Bogdanov. Automated comparison of state-based software models in terms of their language and structure. *ACM Trans. Softw. Eng. Methodol.*, 22(2):13:1–13:37, 2013.

[31] Q. Wang, Y. Brun, and A. Orso. Behavioral execution comparison: Are tests representative of field behavior? In *ICST*, pages 321–332. IEEE Computer Society, 2017.