GK-tail+
An Efficient Approach to Learn Software Models

Leonardo Mariani Member, IEEE, and Mauro Pezzè Senior Member, IEEE, and Mauro Santoro Member, IEEE

Abstract—Inferring models of program behavior from execution samples can provide useful information about a system, also in the increasingly common case of systems that evolve and adapt in their lifetime, and without requiring large developers' effort. Techniques for learning models of program behavior from execution traces shall address conflicting challenges of recall, specificity and performance: They shall generate models that comprehensively represent the system behavior (recall) while limiting the amount of illegal behaviors that may be erroneously accepted by the model (specificity), and should infer the models within a reasonable time budget to process industrial scale systems (performance).

In our early work, we designed GK-tail, an approach that can infer guarded finite state machines that model the behavior of object-oriented programs in terms of sequences of method calls and constraints on the parameter values. GK-tail addresses well two of the three main challenges, since it infers guarded finite state machines with a high level of recall and specificity, but presents severe limitations in terms of performance that reduce its scalability.

In this paper, we present GK-tail+, a new approach to infer guarded finite state machines from execution traces of object-oriented programs. GK-tail+ proposes a new set of inference criteria that represent the core element of the inference process: It largely reduces the inference time of GK-tail while producing guarded finite state machines with a comparable level of recall and specificity. Thus, GK-tail+ advances the preliminary results of GK-tail by addressing all the three main challenges of learning models of program behavior from execution traces.

Index Terms—Dynamic model learning, software models, state based models, guarded finite state machines

1 INTRODUCTION

MODELS play a central role in software engineering, and have been extensively studied to increase the effectiveness and efficiency of technical tasks. Software engineering uses models both defined independently from the code, for example for software specification and design, and derived from the code, for example for program analysis and testing.

Models defined independently from the code are useful, but may be expensive to produce and difficult to maintain while the code evolves. On the contrary, models automatically derived from software systems can be produced with limited human effort, and are perfectly aligned with the implementation.

Models may be extracted from the code either by statically analyzing the source code [1]–[3] or by dynamically analyzing the execution traces [4]–[9]. Models dynamically learned from execution traces can capture dynamic aspects that the static analysis techniques may miss, and suffer less from the presence of infeasible elements than statically inferred models. They find various applications that include specification mining [10]–[12] program comprehension [4], [5], test case generation [7]–[9], [13], fault diagnosis [14], bug fixing [15] and performance evaluation [16].

The many model learning techniques that have been introduced so far can infer different kinds of models, achieve various levels of precision, and provide heterogeneous types of data with diverse applications. Some techniques generate invariants [10]–[12], others transition systems [6], yet others finite state machines [15], [17], [18], message sequence charts [11], [19] or temporal properties [20]. Inferred models have been used to represent a variety of behaviors, including status information, ordering of events, and pre- and post-conditions. So far inference techniques have focused on a specific aspect of the modeled system, and little work has addressed the interplay of the different aspects that characterise complex systems, and that can hardly be captured with a single kind of model.

In this paper, we focus on models dynamically learned from execution traces, and address the problem of learning finite state machines (FSMs) annotated with guard conditions, which integrate information about the ordering of execution of the operations with the conditions on the parameters that govern those operations.

Techniques for learning models of program behavior from execution traces shall address conflicting challenges of recall, specificity and performance: They shall generate models that comprehensively represent the system behavior (recall) while limiting the amount of illegal behaviors that may be erroneously accepted by the model (specificity), and should infer models within a reasonable time budget to process industrial scale systems (performance). The precision of the inferred models in terms of recall and specificity is a paramount property in many application domains, in particular in specification mining, debugging and test generation, where many false negatives (low recall) and many false positives (low specificity) impact on the usefulness of the inferred specifications and on the effectiveness of testing and analysis activities [5], [7], [14]. The performance of the inference process in terms of inference time impacts on the scalability of the approach. Many application domains,
specifically test generation, fault diagnosis and bug fixing, require detailed models that may be quite large already at the class level, and need efficient inference algorithms to scale to industrial size applications [7], [14], [15].

Several approaches address the problem of inferring annotated FSM models from sample traces. Cassel et al. and Aarts et al. exploit active learning techniques [21]–[24]. They incrementally generate the input data for the inference process, and repeatedly check the compliance of the incrementally inferred model with the target software. The many compliance checks heavily impact on the inference costs.

Walkinshaw et al. infer FSMs with classifiers associated with transitions, and constrain the values that can be assigned to the parameters [25]. They infer classifiers on a per label basis, for instance per method, and address well systems where the values of the parameters strongly influence the event sequences.

Other approaches infer guards that characterize the states of the FSM model and do not capture the constraints on the values of the parameters of the transitions [26]–[29]. They require traces with detailed state information, which may not be always easy to mine efficiently.

In previous work, we defined GK-tail, an approach that infers FSMs with transitions associated with conditions that combine information about the ordering of the events and the values of the parameters [6]. The results reported in [6] show that GK-tail addresses well two of the three main challenges, since it infers guarded FSMs with a high level of recall and specificity, but presents severe limitations in terms of performance that can impact on the applicability of the approach to increasingly complex and rapidly evolving systems. GK-tail produces models that seldom overgeneralize the samples available in the traces, but may require several hours to complete, becoming impractical when dealing with large and quickly evolving software systems [30].

GK-tail+ extends the classic k-Tail algorithm [31] to capture sequences of method invocations, and combines it with Daikon [32] to synthesize constraints on parameter values. GK-tail suffers from the large number of Daikon executions, since it invokes Daikon for every method invocation in the traces. This strategy produces a number of Daikon invocations of the order of the number of events that must be processed. Since it is easily possible to collect millions of events even with few executions, the resulting computational cost becomes quickly too high, with a strong impact on the efficiency of GK-tail, thus limiting its applicability to rapidly evolving systems.

In this paper, we present a new approach that maintains the excellent precision and recall of GK-tail while dramatically improving the inference performance. In particular, we define a new algorithm, GK-tail+, that derives an initial finite state machine from traces by considering only the events in the sequences, and limits the invocations of Daikon to the number of transitions in the final model, instead of invoking it for each event in the sequences. Although in the worst case the number of transitions in the model can be of the same order of the number of events in the traces, in practical applications the number of transitions is much smaller than the number of processed events, thus leading to a dramatic improvement on the efficiency of the inference process.

GK-tail+ does not suffer from the performance problems observed when using active learning techniques, since it implements a passive black-box style of learning which does not impose any relevant constraint for its application and does not require compliance checking. GK-tail+ is more suitable than the approach by Walkinshaw et al. for systems where the parameters do not always influence the sequences of events that can be executed next, since it does not exploit constraints in the generalization process, and infers constraints on a per transition basis, discriminating the values that can be assigned to each specific occurrence of an event, for instance each individual occurrence of a same method in the model. GK-tail+ overcomes the performance limitations of GK-tail by proposing a novel process to efficiently infer guards associated to the transitions.

This paper advances the state of the art in model learning by improving the initial results presented in [6] with (i) a new algorithm and two new criteria for generating behavioral models that integrate event sequences and parameter values, (ii) a complete formalization of GK-tail+, and (iii) a set of experimental results that confirm the comparable effectiveness and the dramatic improvement in the efficiency of GK-tail+ with respect to GK-tail.

This paper is organized as follows. Section 2 formally defines GK-tail+, and discusses the main differences with respect to GK-tail. Section 3 presents a set of experimental results that indicate the validity and the limitations of the proposed approaches. Section 4 surveys the related work. Section 5 summarises the main contributions of the paper.

2 GK-TAIL+

GK-tail+ learns guarded finite state machine (gFSM) models of the dynamic behavior of software systems from sets of execution traces. Differently from Extended FSM (EFSM), the parameters used in gFSMs cannot be shared between transitions, that is a parameter in a transition is not visible from other transitions.

Figure 1 illustrates the main steps of the approach. GK-tail+ processes input traces that encode sequences of events annotated with values assigned to the parameters of the events, for example, sequences of method calls annotated with the values of the parameters used in the calls.

Step 1: The merging traces step merges the subsets of the input traces that represent a same scenario, which includes all the traces composed of the same sequence of events with possibly different values of the parameters, into a single generalized trace, which is composed of the same sequence of events of the merged traces, and is annotated with the sets of values associated with the events in the input traces.

The generalized traces correspond to the same scenario represented by the merged traces, but summarize the values that can be assigned to parameters within a single annotation per event. For example, when merging three traces that include an invocation to method setAge(int age) with the parameter values age=15, age=51 and age=28, respectively, the merging traces step produces a generalized trace with a single invocation to the method setAge(int age) annotated with the set {age=15, age=51, age=28} that indicates the values used to invoke setAge(int age) in any of the original traces.
Fig. 1. An overview of GK-tail+

Step 2: The generating the initial FSM step creates an initial FSM shaped as a tree, where each branch of the tree accepts the sequence of events in a generalized trace. The transitions of the initial FSM are labeled with an event and are associated with a set of parameters as the corresponding generalized trace.

Step 3: The merging states step merges the states of the initial FSM that accept the same sequences of operations. Intuitively these states are redundant representations of a same logical state, and thus can be reduced to a single state of the model. The merging states step merges also redundant transitions, which are transitions that start from the same state, end at the same state, and have a same label. Redundant transitions might be created as a consequence of the state merging process. Two redundant transitions are merged into a single transition annotated with the values that annotate the two merged transitions. The resulting annotated FSM is a FSM with transitions annotated with sets of values for the parameters of the represented events.

Step 4: The generating constraints step processes the values that annotate the transitions to generate constraints, which are used as guards for the same transitions. For instance, the generating constraints step may generate the constraint \( \text{age}>0 \) from the values \{age=15, age=51, age=28\} as the guard of a transition that accepts the event \( \text{setAge(int age)} \). The resulting gFSM is a guarded finite state machine, that is, a FSM with transitions annotated with guard conditions.

2.1 Guarded Finite State Machines

In this section we introduce guarded finite state machines (gFSMs), the models that GK-tail+ infers from execution traces to represent the behavior of software systems. gFSM transitions represent the occurrence of the events in the input traces, and are annotated with guards that represent the conditions for their occurrence.
A trace is a sequence of events, each composed of an action and a set of parameters with their values. Actions are operations, and parameters represent the values of the variables associated with an action. For instance, the sequence

\[
\langle\text{setFullName(String name, String surname), name=John surname=Smith}\rangle
\]

\[
\langle\text{setAge(int age), age=18}\rangle
\]

\[
\langle\text{setWeight(int weight), weight=66}\rangle
\]

is a trace with three events that correspond to the method calls: setFullName with two parameters name and surname, and the methods setAge and setWeight with one parameter each, age and weight, respectively. The values associated with the parameters are John, Smith, 18 and 66, respectively.

**Definition 2.1. Trace** Let \( E \) be a finite set of events and \( V \) a finite set of values, a trace \( tr \) is a sequence \( tr = (e_1, . . . , e_n) \) with \( e_i \in E \) and \( v_i \in V \), for \( i = 1 . . . n \). We denote with \( TR \) the set of all the traces.

Since an event might be associated with multiple parameters, \( V \) is a set that includes all combinations of values for any number of parameters. \( V \) might include individual values, such as 18 and 66, but also tuples that can represent the values of multiple parameters. For example the tuple \((\text{John}, \text{Smith})\) can represent the values of two parameters associated with an event in \( E \). While in the formalization we identify the value associated with the parameters by their position abstracting the name of the parameters, in the figures we explicitly indicate the name of the parameters to highlight the correspondence with the parameters in the traces.

Some domains may restrict the pairs \((e_i, v_i)\) to \( V \) to a proper subset. We can cope with these restrictions by simply considering a domain \( D \subseteq E \times V \) and require \( tr_i \in D \) for \( i = 1 . . . n \). Without loss of generality, in the rest of this paper we do not consider such a restriction.

**The right-hand side of Figure 6 shows an example of FSMs whose transitions are annotated with sets of values, indicated with the name of the parameter and the associated value.** In the following, we define FSMs, which are used jointly with an annotation function later in this section, and then guarded FSMs.

**Definition 2.2. Finite State Machine** A finite state machine (FSM) is a tuple \((S, s_0, E, T)\) where:

- \( S \) is a finite set of states
- \( s_0 \in S \) is the initial state
- \( E \) is a finite set of events
- \( T \) is a transition relation \( T : S \times E \rightarrow S \)

FSMs accept traces independently from the values associated with the events. Formally, a FSM accepts a trace \( tr = (e_1, v_1, . . . , e_n, v_n) \) if \( \forall i = 1 . . . n \exists (s_{i-1}, e_i, s_i) \in T \).

Guarded finite state machines extend FSMs with a set of values and by augmenting transitions with conditions on values.

**Definition 2.3. Guarded Finite State Machine** A guarded finite state machine (gFSM) is a tuple \((S, s_0, E, V, G, T)\) where:

- \( S \) is a finite set of states
- \( s_0 \in S \) is the initial state
- \( E \) is a finite set of events
- \( V \) is a finite set of values
- \( G \) is a finite set of guard functions \( g_i : V \rightarrow \{0, 1\} \)
- \( T \) is a transition relation \( T : S \times E \times G \rightarrow S \)

A gFSM accepts a trace \( tr = (e_1, . . . , e_n) \), with \( tr_i = (e_i, v_i) \), such that \( \forall i = 1 . . . n \exists (s_{i-1}, e_i, g_i, s_i) \in T, g_i(v_i) = 1 \).

gFSMs accept only traces with events that match the transition labels and parameter values that satisfy the guards. More formally, a gFSM accepts the set of traces \( \{tr = (e_1, . . . , e_n)\} \), with \( tr_i = (e_i, v_i) \), such that \( \forall i = 1 . . . n \exists (s_{i-1}, e_i, g_i, s_i) \in T, g_i(v_i) = 1 \).

**2.2 Step 1 – Merging traces**

In the merging traces step, GK-tail+ merges single execution traces into generalized traces, which group traces that correspond to the same sequence of events.

GK-tail+ produces generalized traces that associate multiple values with the parameters of a same event, by exploiting Event Equivalence. Two traces are deemed to be event equivalent if they encode the same sequence of events, with possibly different values of the associated parameters. Intuitively, event equivalent traces represent scenarios that have been executed multiple times with possibly different values of the parameters. For example, people who interact...
with a system to create a new account execute sequences of operations that differ in the values entered in the registration form, not in the events in the sequence. \(GK\text{-}tail^+\) merges all these executions into a single generalized trace that encodes the sequence of operations and the values used during the executions of the corresponding traces.

Figure 2 illustrates the input/output behavior of the merging traces step. In the input traces, the events are associated with a value for each parameter. In the output generalized traces, the events are associated with sets of values. In the example, \(GK\text{-}tail^+\) merges the first two and the last two input traces into two generalized output traces, respectively. The events of the generalized traces are associated with sets of values for the parameters.

\(GK\text{-}tail^+\) identifies the traces that must be merged by processing events in the context of their traces, rather than independently from the scenario. It thus merges the data associated with the calls of a same method in a set of event equivalent sequences, and merges differently the calls to the same method in the context of a different set of event equivalent sequences. For example, the data associated with \(\text{setAge}(\text{int age})\) might be values greater than 50 when invoked in a context of retired people, and simply natural numbers in other scenarios. Merging the data without considering the different scenarios would lead to the same set of values and eventually a same constraint for \(\text{setAge}(\text{int age})\), thus missing the interesting information about the different set of data exchanged in the different scenarios.

Merging single traces into generalized traces improves the overall performance of the approach, by reducing the amount of traces to be processed in the following steps.

**Definition 2.4. Event Equivalent Traces**

Given \(tr = \langle tr_1, \ldots, tr_n \rangle\) with \(tr_i = (e_i, v_i)\) and \(tr' = \langle tr'_1, \ldots, tr'_n' \rangle\) with \(tr'_i = (e'_i, v'_i)\), we say that \(tr\) is event equivalent to \(tr'\) if:

- \(n = n'\), and
- \(e_i = e'_i\) for \(i = 1 \ldots n\)

**Event Equivalence** is a reflexive, symmetric and transitive binary relation, and thus is an equivalence relation that partitions traces in equivalent classes. Given a set \(TR\) of traces, we denote the set of equivalent classes with \(TR/\sim\) and the elements of this set with \(\{tr\}\).

The merging traces step computes a generalized trace for each element in \(TR/\sim\), that is, it turns a set of event equivalent traces into a generalized trace.

The events of a generalized trace are associated with multisets of values, where a multiset is a set that may contain multiple instances of the same element. A multiset is a pair \((A, m)\), where \(A\) is a set and \(m : A \rightarrow \mathbb{N}^+\) is a function that returns the cardinality of each element. The cardinality of an element is 1 if the element occurs once, 2 if the element occurs twice, and so on. Given a set \(S\), we denote with \(\mathbb{M}(S)\) the multiset \((S, m)\).

**Definition 2.5. Generalized Traces**

A generalized trace is a trace \(gt = \langle gt_1, \ldots, gt_n \rangle\), where \(gt_i = (ev_i, w_i)\) with \(ev_i \in E\) and \(w_i \in \mathbb{M}(V)\). We indicate with \(GT\) the set of all the generalized traces.

Given an input set \(TR\) of traces for each \([tr] \in TR/\sim\), the merging traces step produces a generalized trace \(gt = \langle gt_1, \ldots, gt_n \rangle \in GT\) with \(gt_i = (ev_i, w_i)\) as follows:

\[
ev_i = ev_i, \text{ for any } \langle (ev_1, v_1), \ldots, (ev_n, v_n) \rangle \in [tr] \\
w_i = \bigcup_{(ev_i, v_i) \in [tr]} v_i
\]

**2.3 Step 2 – Generating The Initial FSM**

In the generating the initial FSM step, \(GK\text{-}tail^+\) produces an annotated FSM by simply merging the set of generalized traces produced in the former step into a tree shaped FSM, with an annotation function that keeps track of the values associated with the events in the input generalized traces. The initial FSM accepts all and only the input generalized traces. Differently from the classic state merging processes [31], [33], [34], \(GK\text{-}tail^+\) does not merge the common prefixes of the branches in the tree, but merges operations later in the process (Step 3).

Figure 3 illustrates the input/output behavior of this step. In the output FSM each branch of the tree corresponds to an input generalized trace.

More formally, given a set of \(m\) generalized traces \(gt^j = \langle gt^j_1, \ldots, gt^j_n \rangle, j = 1 \ldots m\) with \(gt^j_i = (ev^j_i, w^j_i)\), \(GK\text{-}tail^+\) generates a FSM \((S, s_0, E, T)\) and an annotation function \(A : T \rightarrow \mathbb{M}(V)\) defined as follows:

- \(S = \{s_0, \ldots, s_l\}\), where \(l = \sum_{j=1}^m n_j + 1\) is a set of states
- \(s_0\) is the initial state
- \(E = \bigcup_{j=1}^m ev^j_i\) is the set of event symbols
2.4 Step 3 – Merging States

In the merging states step, GK-tail+ generalizes the initial FSM into an FSM that accepts a larger set of events, to better reflect the behavior of the monitored software system.

GK-tail+ generalizes the initial FSM by exploiting a notion of observationally equivalent states: two states are observationally equivalent if they accept the same set of behaviors. When the FSM model is complete, observationally equivalent states are redundant representations of a same program state, and the model can be safely simplified into a more compact model by merging the observationally equivalent states.

Since the input traces represents a partial sample of the system behavior, although the initial FSM might include multiple redundant representations of equivalent program states, these potentially redundant states may not be observationally equivalent because they might accept different subsets of the full set of event sequences that they should accept. Merging such potentially redundant states produces a more compact and useful model, which may overgeneralize the system behavior.

GK-tail+ merges states that are observationally equivalent with respect to bounded sequences of events, following the approach of the well known KTail algorithm [31].

The set of the event sequences with a maximum length $k$ accepted by a state $s$ is called the $k$Future of $s$, also denoted as $k$Future$(s)$, and is defined as follows.

![Diagram](https://example.com/diagram.png)

**Fig. 4. Step 3 - Merging States**

- $T(s_a, e) = s_{a+1}$ iff $\exists i, k$ s.t.
  - $a = \sum_{j=1}^{i-1} n_j + k$, $a$ represents the position of the $k^{th}$ symbol in the $i^{th}$ generalized trace
  - $e = ev_k^i$
- $V$ is a set of tuples of arbitrary size, and with values in $\mathbb{R} \cup \text{String}$, $V$ represents the set of variable values
- $A$ is an annotation function that associates the transitions with multisets of values in $V$ and $A(t) = v$, iff $event_w(t) = v$, where
  - $v \in V$
  - $t \in T$
  - $event(t)$ indicates the item $(ev_k^i, w_k)$ in the generalized trace that corresponds to $t$
  - $event_w(t)$ and $event_w(t)$ indicate the event and the set of parameter values, respectively

**Definition 2.6. $k$Future**

Given an FSM $= (S, s_0, E, T)$, $s \in S$, and $k \in \mathbb{N}$, $k$Future$(s)$ is the set of all the sequences $\{seq_1, \ldots , seq_n\}$, s.t., $seq_i = \{e_1, \ldots , e_n\}$, $n_i \leq k$ and $\exists (s_j, e_j, s_j') \in T$, with $j = [0, \ldots , n_i]$ and $s_0 = s$.

To further accommodate model incompleteness, GK-tail+ considers two criteria for comparing kFutures. The Event Equivalence criterion merges two states if their kFutures are the same. The Event Subsumption criterion merges two states if the kFuture of one state includes the kFuture of the other state. The Event Subsumption criterion tolerates a higher degree of incompleteness in the model than the Event Equivalence criterion. In fact, it is enough that one of the redundant states has been well covered in the traces, to merge the well-covered state with all the other redundant and partially covered representations of the same state, if any. This is not possible with Event Equivalence that only merges states that accept exactly the same behaviors.

**Definition 2.7. Event Equivalence**

Given an FSM $= (S, s_0, E, T)$, two states $s_1, s_2 \in S$, and $k \in \mathbb{N}$, we say that $s_1$ is event equivalent to $s_2$, indicated as $s_1 =_{eq} s_2$, iff $k$Future$(s_1) = k$Future$(s_2)$.

The Event Equivalence criterion merges two states $s_1$ and $s_2$ iff $s_1 =_{eq} s_2$.

**Definition 2.8. Event Subsumption**

Given an FSM $= (S, s_0, E, T)$, two states $s_1, s_2 \in S$, and $k \in \mathbb{N}$, we say that $s_1$ is event subsumed by $s_2$, indicated as $s_1 \subseteq s_2$, iff $k$Future$(s_1) \subseteq k$Future$(s_2)$.

The Event Subsumption criterion merges two states $s_1$ and $s_2$ iff $s_1 \subseteq s_2$ or $s_2 \subseteq s_1$.

In the merging state step, GK-tail+ substitutes two states that are equivalent according to the chosen criterion with a merged state, and replaces the input/output transitions of the deleted states with new input/output transitions of the merged state. GK-tail+ obtains the new transitions by substituting the deleted states with the merged state. This process may produce pairs of similar transitions, that is, transitions that connect the same pair of states and share the same label. GK-tail+ removes one of two similar transitions, merges the annotation of the removed transition with the annotation of the remaining transition, and updates the annotation function accordingly.

GK-tail+ merges states according to the selected criterion iteratively until there are no more states that can be merged,
Fig. 5. Comparing Event Equivalence and Event Subsumption merging criteria

without enforcing any specific order on the comparison of the states. The resulting FSM accepts all the input traces.

Figure 4 illustrates the input/output behavior of the merging states step. In the simple example shown in the figure, GK-tail+ produces the same FSM with either of the two merging criteria, however in general GK-tail+ produces different FSMs depending on the chosen criterion.

Figure 5 shows an example of different FSMs produced with the two merging criteria. The top of the figure shows an excerpt of a FSM obtained after few iterations of the state merging process, while the bottom of the figure presents the different FSMs produced from the top excerpt with the two merging criteria.

The kFuture of state 10 in the FSM at the top of Figure 5 strictly includes the kFuture of state 1: kFuture(1) = \{write, write, closeFile\} and kFuture(10) = \{write, write, closeFile\} \{write, closeFile\}. The different kFuture of the two states may easily depend on the limited amount of input traces: In state 1, the system has been executed only with a sequence of two write operations followed by a file close operation, while in state 10, the system has also executed both with the former sequence of operations and with a sequence of a write operation followed by a file close operation. A richer set of traces might have produced the same kFuture for both states.

States 1 and 10 cannot be merged according to the Event Equivalence criterion because kFuture(1) \neq kFuture(10), but they can be merged according to the Event Subsumption criterion because kFuture(1) \subset kFuture(10). This example illustrates the different flexibility of the two criteria, when traces are largely incomplete, as often happen for large systems.

The FSM obtained with Event Equivalence contains more states but accepts less behaviors than the FSM obtained with Event Subsumption. While this overgeneralization may introduce some spurious sequences, in the case of a relatively sparse sampling of the execution space, it may capture many legal behaviors of the monitored system. For example, both criteria generate a self-loop transition, but the self-loop transition of the Event Subsumption FSM captures many legal behaviors of the system that the Event Equivalence FSM does not accept.

We discuss in details the usefulness of the two criteria in Section 3 when we present the results of a large set of experiments.

2.5 Step 4 – Generating Constraints

In the generating constraints step, GK-tail+ produces a gFSM from the FSM and the annotation function produced in the merging states step by substituting the values that the annotation function associates with the transitions of the input FSM with constraints associated with the produced gFSM transitions.

Figure 6 illustrates the input/output behavior of the generating constraints step. GK-tail+ generates the constraints associated with the output gFSM by using the Daikon invariant detection technique [32].

Given a set A of (variable, value) pairs, we indicate the constraints that Daikon generate from A with Daikon(A(t)). GK-tail+ invokes Daikon for each transition to synthesize a
constraint from the values associated with the transition in
the input FSM. The constraints that Daikon generates are
statistically relevant Boolean expressions that accept all the
values provided as input.

More formally, given an FSM \((S, s_0, E, T)\) and an annotation
function \(\hat{A} : T \rightarrow M(V)\), the generating constraints
step produces a \(g\)FSM \(G_S M = (S', s'_0, E', V', G', T')\) de-
efined as follows:

- \(S' = S\)
- \(s'_0 = s_0\)
- \(E' = E\)
- \(V' = V\)
- \(\forall t \in T, t = (s_a, e, s_b), \exists t' \in T', t' = (s_a, e, g, s_b), \) with
- \(g = Daikon(A(t))\)
- \(G' = \bigcup_{(s_a, e, g, s_b) \in T'} g\)

2.6 Delayed constraint computation

\(GK-tail+\) largely improves over the original \(GK-tail\) approach
thanks to the delayed computation of the constraints.

While \(GK-tail\) generates constraints early in the process
for each element of the generalized traces, \(GK-tail+\) synthe-
sises constraints late in the process for the values associated
with the transitions of the FSM after merging the states of
the initial FSM. Computing constraints after merging states
cannot both dramatically reduce the expensive computation of
constraints that limit the scalability of \(GK-tail\) and produce a
model that better approximates the behavior of the original
system, especially when dealing with a limited set of samples.

\(GK-tail\) associates constraints with traces by running
Daikon on the generalized traces. More formally, given a
generalized trace \(gt = (gt_1, \ldots, gt_n)\), with \(gt_i = (ev_i, w_i)\),
\(GK-tail\) runs Daikon on each set of samples \(w_i\) and generates
traces enriched with constraints where each item of the
trace is a pair \((ev_i, c_i)\), with \(c_i = Daikon(w_i)\). As discussed
earlier in this section, \(GK-tail+\) delays the computation of
constraints after the merging states step.

By computing constraints early over generalized traces
rather than later over the annotated FSM, \(GK-tail\) (i) requires
many invocations of Daikon, one for each element of each
trace, while \(GK-tail+\) requires far less invocations of Daikon
(ii) generates constraints from smaller data sets than \(GK-
tail+\) and consequently generates constraints that more likely
overfit the observations than \(GK-tail+\), (iii) adopts a more
complex and expensive state merging process than \(GK-
tail+\), because comparing the kFuture of two states requires
comparing both event sequences and constraints.

In the next section we present empirical results that
confirm our hypotheses about the improved efficiency of
\(GK-tail+\) with respect to \(GK-tail\).

2.7 Convergence

\(GK-tail+\) shares several properties with k-Tail [31]. When
applied to a set of traces produced from a given FSM, both
\(GK-tail+\) and k-Tail may not converge to the original FSM,
and may produce an imprecise FSM with respect to both
completeness and soundness, that is, the FSM produced
with \(GK-tail+\) may accept traces that the original FSM rejects,
and may reject traces that the original FSM accepts.

Since we do not have control over the traces, which
correspond to a set of executions, the lack of convergence
is not a problem as long as the inferred models are of
good quality in the practical cases, that is, they capture well
the behavior of the analyzed program, as confirmed by the
experimental results reported in Section 3.

\(GK-tail+\) infers models that accept all the traces used
for the inference. This could be easily demonstrated by
analysing the inference process.

In the first step, \(GK-tail+\) merges the (event equivalent)
input traces into a set of generalized traces. Since the
merging process associates multiple parameter values to
each event without dropping any sequence of events, the
set of generalized traces includes all the executions that
correspond to the input traces.

In the second step, \(GK-tail+\) generates an initial FSM that
accepts exactly all and only the executions represented with
the generalized traces. In fact each trace is mapped to a
branch of the FSM and each set of parameter values asso-
ciated with an event is used to annotate the corresponding
transition.

In the third step, \(GK-tail+\) iteratively merges states. The
FSM obtained by merging two states accepts a set of traces
that contains all the traces accepted by the FSM before the
merging. This condition holds when looking both at the
events, like in the k-Tail algorithm, and at the parameter
values, which are not dropped during the state merging
process.
In the fourth step, GK-tail+ uses Daikon to generate guards from the sets of parameter values associated with the transitions. Since Daikon always generates constraints that accept all the input samples, the guards associated with the transitions are guaranteed to accept all the samples that annotate the same transitions. If Daikon generates no constraint for a set of parameter values, the corresponding transition is associated with no guard, and accepts every possible value of the parameters, including the ones that annotate the same transition. Thus a model generated with GK-tail+ accepts all the input samples by construction.

### 3 Experimental Evaluation

GK-tail+ builds on top of GK-Tail by defining and formalizing new algorithms and criteria for generating behavioral models that integrate event sequences and parameter values aiming to overcome the performance limitations of the original GK-Tail approach.

In this section we present the results of a comparative evaluation of GK-tail+ and Gk-tail, which shows that GK-tail+ can indeed generate models of the same quality as Gk-tail in less than half of the time, thus confirming the progresses in terms of applicability and scalability fostered by the new algorithms and criteria that characterize GK-tail+.

Section 3.1 introduces the setup of our empirical evaluation. Section 3.2 discusses the recall, specificity and balanced classification rate metrics that we use to evaluate the inference algorithms. Section 3.3 presents the results of an initial experiment for tuning the parameter $k$ for both GK-tail+ and Gk-tail, Sections 3.4, 3.5 and 3.6 discuss the quality of the inferred models in terms of recall, specificity and balanced classification rate (BCR), respectively. Section 3.7 empirically evaluates the degree of dependence of the algorithms from the amount of traces used to generate the models. Section 3.8 provides comparative data about the performance of GK-tail+ and Gk-tail. Section 3.9 discusses the main threats to the validity of the results presented in the paper.

#### 3.1 Empirical Setup

We evaluate GK-tail+ comparatively with Gk-tail to assess both the quality of the inferred gFSMs and the cost of the inference process. In our comparative evaluation, we consider the full range of inference criteria defined for both techniques: Equivalence, Weak Subsumption and Strong Subsumption for Gk-tail, Event Equivalence and Event Subsumption for GK-tail+.

Both GK-Tail and GK-tail+ produce generalized traces to infer gFSMs, but Gk-tail produces guards before building the initial FSM, by running Daikon on the samples associated with every event, while GK-tail+ merges states before generating constraints, and runs Daikon fewer times on larger sets of samples. Thus, transitions are associated with guards in the initial FSM generated by Gk-tail, while transitions are associated with values only in the initial FSM generated by GK-tail+.

The Gk-Tail criteria merge states by comparing their kFuture, which includes both events and guards. The Equivalence criterion requires both the same events and guards in the kFuture, the Weak Subsumption criterion requires the same events, and the guards of one sequence be included in the guards of the other sequence in the kFuture, and the Strong Subsumption criterion requires both events and guards of one sequence be included in the events and guards of the other sequence in the kFuture.
interested in a detailed description of the GK-Tail criteria can refer to [6]. As discussed in the former sections of this paper, the GK-tail+ Event Equivalence and Event Subsumption criteria merge states with the same or included events in the kFuture, respectively.

We designed our empirical study referring to the practical situation of developers who want to infer models of the available artifacts, without paying the extra effort of implementing additional test cases and modifying the application to ease the model inference task. For this reason, we selected a set of subject classes from widely used applications, and used the original test suites distributed with these applications to produce the input traces.

We generated models of the behavior of the subject classes by considering the calls to the methods implemented with these applications to produce the input traces.

TABLE 2

<table>
<thead>
<tr>
<th>Subject Class</th>
<th>GK-tail</th>
<th>GK-tail+</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Equivalence</td>
<td>Weak</td>
</tr>
<tr>
<td></td>
<td>CC States Transitions</td>
<td>CC States Transitions</td>
</tr>
<tr>
<td>Guava ArrayListMultimap</td>
<td>425 975 1398</td>
<td>425 844 1267</td>
</tr>
<tr>
<td>HashMultiset</td>
<td>203 422 623</td>
<td>273 386 657</td>
</tr>
<tr>
<td>HashBMap</td>
<td>276 840 1214</td>
<td>659 673 1330</td>
</tr>
<tr>
<td>HashMultimap</td>
<td>208 430 636</td>
<td>224 406 628</td>
</tr>
<tr>
<td>HashMultiset</td>
<td>142 222 362</td>
<td>142 221 361</td>
</tr>
<tr>
<td>ImmutableBiMap</td>
<td>136 286 94</td>
<td>46 92 136</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>24 74 96</td>
<td>28 67 93</td>
</tr>
<tr>
<td>ImmutableMutiset</td>
<td>202 418 618</td>
<td>210 402 610</td>
</tr>
<tr>
<td>LinkedHashMultimap</td>
<td>476 1057 1531</td>
<td>502 942 1442</td>
</tr>
<tr>
<td>LinkedHashMultiset</td>
<td>242 356 596</td>
<td>244 349 591</td>
</tr>
<tr>
<td>LinkedListMultimap</td>
<td>382 1196 1576</td>
<td>411 1087 1496</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>695 1579 2272</td>
<td>777 1431 2206</td>
</tr>
<tr>
<td>TreeMultiset</td>
<td>840 716 1194</td>
<td>891 500 1389</td>
</tr>
</tbody>
</table>

Legend

Columns Subject Class indicates the name of the class used in the evaluation.
Columns GK-tail and GK-tail+ indicate the inference algorithm used to generate the FSMs.
Columns Equivalence, Weak, Strong, Event Equivalence and Event Subsumption indicate the specific criterions used to infer the gFSMs. Columns CC, States and Transitions indicate the cyclomatic complexity, the number of states, and the number of transitions of the model inferred with $k = 2$, respectively.

Table 1 reports some metrics that size the source code and the traces analyzed in the experiments. The table indicates the size and complexity of the subject classes in terms of lines of code (Locs) and weighted methods per class (WMC), and quantifies the size and complexity of the behavioral information processed by the algorithms in terms of number of traces (Number), total number of events (Total Length) and number of distinct events in the traces (Alphabet). Table 2 provides data about the size and the structure of the inferred models. The table shows the cyclomatic complexity (CC), the number of states (States), and the number of transitions (Transitions) of the models generated with the different inference criteria (Equivalence, Weak, Strong, Event Equivalence, Event Subsumption) of GK-tail and GK-tail+. Since the FSMs are connected graphs, we used the standard formula $\#\text{transitions} - \#\text{states} + 2$ to compute their cyclomatic complexity. The size and complexity of both the classes and the models vary a lot, for example the number of Locs per class ranges from 22 to 696, and the number of states ranges from 51 to 1579, indicating the variety of situations faced in our study. All the classes produce models of non-negligible size and complexity, with some highly challenging cases.

In Table 2 we highlight the smallest value of the different metrics (cyclomatic complexity, number of states and number of transitions) for each class in bold. In the vast majority of the cases, the Event Subsumption criterion of

1. The TPTP project is not active anymore. However many other monitoring solutions, such as AspectJ [35] and BCEL [36], can be used to collect method invocations producing the same result.
3.2 Metrics

We compare GK-tail+ with GK-tail in terms of recall, specificity and Balanced Classification Rate (BCR) of the inferred models.

The recall measures the completeness of the inferred models, and is defined as the fraction of traces accepted by an inferred model with respect to the number of traces used to infer the model [39]. More formally, given a model $M$ for a program $P$ and a set of legal traces $T$ obtained by executing $P$, the recall of $M$ is:

$$\text{recall}(M) = \frac{\text{number of traces that } M \text{ accepts}}{\text{number of traces in } T}$$

We produced the traces for this study by executing the subject programs with the test cases distributed with the programs themselves, and recording the sequences of method calls and parameter values. We generated the gFSMs for each subject program using both the two inference criteria implemented in GK-tail+ and the three criteria implemented in GK-tail. We computed the recall of the models generated for a program $P$ with the $n$-fold cross-validation process [40], which consists in partitioning the set of traces obtained by executing $P$ into $n$ sets of the same size, and using $n - 1$ sets for inferring the model (training) and the remaining set to compute the number of legal traces accepted by the model (validation). This process is repeated $n$ times, each time using a different set for the validation phase. The recall is computed as the average of the $n$ values collected with this process. We use $n = 10$ in all the experiments unless differently indicated.

The specificity measures the ability of the inferred gFSMs to reject illegal behaviors, that is behaviors that do not correspond to legal execution of the program $P$, and is defined as the fraction of illegal traces that are correctly rejected by the inferred gFSMs [39]. More formally, given a model $M$ for a program $P$ inferred with a set of legal traces $T$ and a set of illegal traces $I \mid I \cap T = \emptyset$, the specificity of $M$ is:

$$\text{specificity}(M) = \frac{\text{number of traces in } I \text{ that } M \text{ rejects}}{\text{number of traces in } I}$$

We computed the specificity of each gFSM by feeding the gFSM with illegal traces and calculating the fraction of illegal traces that GK-Tail and GK-tail+ correctly reject. We generated illegal behaviors by permuting the correct traces used to infer the gFSMs with three operators, swap, r-swap and del, and keeping only the illegal traces.

Given an interaction trace $(t_{r_1} \ldots t_{r_n})$, where each element $t_{r_i}$ is a pair $(e_i, v_i)$ composed of an event $e_i$ and a set of values $v_i$ associated with the event, the operators work on randomly selected elements. The swap operator swaps two consecutive elements $t_{r_i}$ and $t_{r_{i+1}}$, the r-swap operator swaps two non consecutive elements $t_{r_i}$ and $t_{r_j}$, where $j > i + 1$, and the del operator removes an element $t_{r_i}$ from the interaction trace. Since traces represent sequences of method calls produced by executing the body of the methods in the classes under test, changing the order of events or removing events are likely to produce illegal traces, that is, traces that do not correspond to the control flow of the program under test.

Changing the order of events or removing events may produce legal traces by chance. To reduce the impact of legal traces produced by chance, we discard traces that match a trace in the set of traces used to infer the model. In this way, we reject trivial mutations that correspond to legal traces. This has not happened frequently, but improved the value of the estimated specificity.

We assessed the suitability of the sets of illegal traces to compute valid values for the specificity of the models, by computing both the Clopper-Pearson and the Agresti-Coull confidence intervals [41], [42], which work well for binomial confidence intervals, that correspond to our case. We considered two confidence intervals to mitigate the bias that might be introduced by using only one criterion. For each operator and subject class, we continue generating illegal traces until the computed specificity value reached a 95% confidence interval with an error in the range ±0.03 for both the Clopper-Pearson and the Agresti-Coull confidence intervals.

Finally, we also compare the effectiveness of GK-tail+ and GK-tail in terms of both recall and specificity with the Balanced Classification Rate (BCR), which is defined as the average value of recall and specificity.

3.3 Parameter Tuning

The state merging process in both GK-tail and GK-tail+ may depend on the value of the parameter $k$ that determines the maximum length of the event sequences in the kFuture of a state. It is well-known that small values of $k$ are needed to achieve a reasonable generalization of the behavior represented with the input traces, and in many studies, the most common values for $k$ are 2 or 3 [14], [30], [33], [43]. In this subsection we report the results of an empirical study about the impact of $k$ on the GK-tail+ and GK-Tail approaches, to determine the value of $k$ that we used in our empirical investigation.

We compare the models inferred with the five criteria considered in this evaluation (the old three criteria for GK-tail and two new criteria for GK-tail+) considering the values 1, 2, 3 and 4 for $k$, leading to a total of 20 configurations investigated for each class. We conducted exhaustive experiments with the 20 configurations on five classes of various sizes and complexity, preferring classes that can be assessed quickly: the classes HashMultiset, HashMutimap, ImmutablesListMultimap and TreeMultiset from the Guava library, and the class Duration from the Joda-Time library. We compare the models obtained with the different
values of \( k \) in terms of size and BCR of the inferred models, and inference time.

Table 3 reports the results. Columns *States* indicate the average number of states in the models inferred with each criterion. They also indicate the reduction in the number of states in percentage with respect to model generated with *GK-tail* executed with the *Equivalence* criterion, which produces the largest models. Columns *BCR* indicate the average BCR of the inferred models. Low BCR values characterize only the classes chosen for this experiment and are not representative of the general results reported in Section 3.6. Columns *Time* indicate the time required for the inference process in seconds as the average among the runs for each criterion. The number in parentheses indicates the number of timeouts that we set to 6 hours.

We can observe that the size of the model grows with \( k \), while the degree of reduction does not depend significantly on the criterion. The dependency on \( k \) confirms the intuition that larger values of \( k \) that identify larger kFutures thus reduce the number of states that can be merged. The magnitude of this phenomenon is significant: The models for \( k = 4 \) and \( k = 3 \) are 2 and 1.5 times larger that the models for \( k = 2 \), respectively.

The quality of the models inferred with \( k = 4 \) is slightly higher than the models inferred with lower values of \( k \), but their inference time is an order of magnitude higher than the inference time of the other configurations. Moreover the configuration with \( k = 4 \) is the only configuration that experiences several timeouts, indicating a lack of practical applicability of configurations with \( k = 4 \).

The configuration with \( k = 1 \) is the fastest to compute, but it is showing a non-negligible degradation of the quality of the models compared to the models obtained with \( k = 2 \) and \( k = 3 \). The configurations with \( k = 2 \) and \( k = 3 \) are characterized by comparable time and BCR, with the configuration with \( k = 2 \) producing smaller models than the one with \( k = 3 \). Since conducting all the experiments with both configurations would be infeasible, and since the models produced with \( k = 2 \) and \( k = 3 \) are comparable, we conducted all the experiments only with the \( k = 2 \) configuration that generates smaller and potentially most practical models than the \( k = 3 \) configuration.

### 3.4 Recall

Table 4 presents the results of the experimental comparison of the recall rate of *GK-tail+* and *GK-tail*. Columns *Recall* report the recall of the models generated with the five inference criteria for the subject classes. Since the accuracy of the inferred models depends on the amount of input traces, we sorted the subject classes by the number of available traces, and computed the median both for all and for the top ten classes.

Both *GK-tail+* and *GK-tail* produce high-quality models when the amount of available traces is large (recall > 80% for the top ten classes in the table), and low-quality models when the amount of available traces is low (recall < 41% for the bottom ten classes.) This result intuitively confirms the dependency of the quality of the models on the extensiveness of the executions used for the inference. This observation is further confirmed by comparing the ratio of the traces used in the inference with the size of the classes shown in Table 1. Such ratio is significantly higher for the top ten classes than for the bottom ten classes, suggesting that extensively covering the behaviors of a class may largely impact on the quality of the inferred models.

Differently from other approaches that infer FSMs, both *GK-tail* and *GK-tail+* produce FSMs augmented with guards (gFSMs). Guards improve the expressive power of the models, but may reduce the accuracy of the model that may reject a large amount of legal behaviors. We evaluated this phenomenon by measuring the percentage of legal traces that are rejected by the inferred models due to inaccurate guards. Column *Traces Rejected due to Guards* of Table 4 reports the results highlighting in bold the values higher than 5%. The number of traces rejected due to inaccurate guards is always small and only in few cases higher than 5%, especially when considering the models generated with a set of traces that sample well the program execution space.

The results show that adding guards to FSMs has a negligible negative effect on recall, which is largely compensated by the reduction of false positives, that is the ability of rejecting traces with illegal data values. This capability is unique of models with guards, and is outside the capability of classic FSMs, including the one inferred with k-Tail.

Figure 7 visualizes the data reported in Table 4 in the
TABLE 4
Comparative evaluation of the recall of GK-tail+ and GK-tail

<table>
<thead>
<tr>
<th>Subject Class</th>
<th>Traces</th>
<th>Recall (10-fold cross-validation)</th>
<th>Traces Rejected Due to Guards</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Equivalence</td>
<td>Weak</td>
</tr>
<tr>
<td>HashBiMap</td>
<td>1430</td>
<td>97.97%</td>
<td>97.97%</td>
</tr>
<tr>
<td>ArrayListMultimap</td>
<td>900</td>
<td>91.20%</td>
<td>92.00%</td>
</tr>
<tr>
<td>LinkedHashSetMultimap</td>
<td>850</td>
<td>89.34%</td>
<td>89.81%</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>86.92%</td>
<td>87.04%</td>
<td>88.22%</td>
</tr>
<tr>
<td>ImmutableListMultiset</td>
<td>620</td>
<td>82.09%</td>
<td>82.09%</td>
</tr>
<tr>
<td>ConcurrentHashSetMultiset</td>
<td>620</td>
<td>95.40%</td>
<td>95.40%</td>
</tr>
<tr>
<td>LinkedHashSetMultimap</td>
<td>360</td>
<td>80.78%</td>
<td>81.33%</td>
</tr>
<tr>
<td>ImmutableBiMap</td>
<td>310</td>
<td>97.74%</td>
<td>97.74%</td>
</tr>
<tr>
<td>LinkedHashSetMultiset</td>
<td>260</td>
<td>84.13%</td>
<td>84.13%</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>220</td>
<td>97.73%</td>
<td>97.73%</td>
</tr>
</tbody>
</table>

Legend
Column **Subject Class** identifies the case studies.
Column **Traces** reports the number of traces collected for the subject class.
Column **Recall (10-fold cross-validation)** presents the recall values obtained with the different inference criteria with the 10-fold cross-validation process.
Column **Traces Rejected Due to Guards** indicates the number of traces that have been rejected due to the presence of an inaccurate guard condition in the inferred model.

The box plots in Figures 8 and 9 visually present the results according to the permutation operators; Tables 5, 6 and 7 analytically report the values obtained with the swap, r-swap, and del operators, respectively. Each table indicates the subject class, the number of illegal traces that have been used to compute specificity (columns **Min** and **Max**), and the specificity for the five criteria. Since the confidence depends on both the number of traces and the specific criterion, the number of illegal traces necessary to reach 95% confidence varies case by case. For this reason column trace reports the minimum and maximum number of traces that have been used to compute the specificity for the different criteria. Each table is split in two parts. The top part reports the results for the models generated with a good number of traces, while the bottom part reports the results for the models generated from few traces, consistently with the previous tables.

The results show that the specificity of the models generated with the **Event Equivalence** criterion is consistent with the specificity of the models generated with **Equivalence** and

In summary, enriching the model with guards improves the expressiveness of the models with little impact on the recall.

3.5 **Specificity**

The values of recall that we measured for **GK-tail** and **GK-tail+** indicate that the inferred qFSMs accept a large amount of correct behaviors, and thus are good candidates for approximating the system behavior. In this section we measure the **specificity** of the models inferred with **GK-tail** and **GK-tail+**.

The box plots in Figures 8 and 9 visually present the results according to the permutation operators; Tables 5, 6 and 7 analytically report the values obtained with the swap, r-swap, and del operators, respectively. Each table indicates the subject class, the number of illegal traces that have been used to compute specificity (columns **Min** and **Max**), and the specificity for the five criteria. Since the confidence depends on both the number of traces and the specific criterion, the number of illegal traces necessary to reach 95% confidence varies case by case. For this reason column trace reports the minimum and maximum number of traces that have been used to compute the specificity for the different criteria. Each table is split in two parts. The top part reports the results for the models generated with a good number of traces, while the bottom part reports the results for the models generated from few traces, consistently with the previous tables.

The results show that the specificity of the models generated with the **Event Equivalence** criterion is consistent with the specificity of the models generated with **Equivalence** and
### TABLE 5
Comparative evaluation of the specificity of the inference criteria of GK-tail and GK-tail+ for the swap permutation operator

<table>
<thead>
<tr>
<th>Subject Class</th>
<th>Min</th>
<th>Max</th>
<th>Equivalence</th>
<th>Weak</th>
<th>Strong</th>
<th>Event Equivalence</th>
<th>Event Subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashBiMap</td>
<td>662</td>
<td>1099</td>
<td>84.29%</td>
<td>81.50%</td>
<td>57.63%</td>
<td>81.50%</td>
<td>54.23%</td>
</tr>
<tr>
<td>ArrayListMultimap</td>
<td>227</td>
<td>491</td>
<td>97.80%</td>
<td>97.80%</td>
<td>90.79%</td>
<td>94.85%</td>
<td>90.02%</td>
</tr>
<tr>
<td>LinkedHashMultimap</td>
<td>187</td>
<td>444</td>
<td>98.93%</td>
<td>98.93%</td>
<td>94.10%</td>
<td>98.93%</td>
<td>91.44%</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>202</td>
<td>373</td>
<td>98.51%</td>
<td>98.51%</td>
<td>95.47%</td>
<td>98.51%</td>
<td>93.57%</td>
</tr>
<tr>
<td>ImmutableListMultiset</td>
<td>292</td>
<td>762</td>
<td>99.66%</td>
<td>99.66%</td>
<td>83.85%</td>
<td>99.66%</td>
<td>80.45%</td>
</tr>
<tr>
<td>ConcurrentHashMultiset</td>
<td>249</td>
<td>609</td>
<td>100.00%</td>
<td>100.00%</td>
<td>96.14%</td>
<td>97.19%</td>
<td>86.21%</td>
</tr>
<tr>
<td>LinkedListMultiset</td>
<td>171</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>98.14%</td>
<td>99.42%</td>
<td>98.51%</td>
</tr>
<tr>
<td>LinkedListHashMultiset</td>
<td>277</td>
<td>721</td>
<td>96.39%</td>
<td>95.90%</td>
<td>89.66%</td>
<td>100.00%</td>
<td>89.43%</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>499</td>
<td>1099</td>
<td>100.00%</td>
<td>100.00%</td>
<td>93.32%</td>
<td>95.68%</td>
<td>82.11%</td>
</tr>
</tbody>
</table>

**Legend**
Columns *Traces, Min, Max* reports the number of traces, from smallest to largest, fed to the subject class.
Columns *Equivalence, Weak, Strong, Event Equivalence and Event Subsumption* identify the inference criteria.
Column *Subject Class* identifies the case studies.
Column *swap* identifies the permutation operator.
Columns *Specificity* presents the specificity values obtained with the different inference criteria.

### TABLE 6
Comparative evaluation of the specificity of the inference criteria of GK-tail and GK-tail+ for the r-swap permutation operator

<table>
<thead>
<tr>
<th>Subject Class</th>
<th>Min</th>
<th>Max</th>
<th>Equivalence</th>
<th>Weak</th>
<th>Strong</th>
<th>Event Equivalence</th>
<th>Event Subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashBiMap</td>
<td>1024</td>
<td>1095</td>
<td>59.55%</td>
<td>56.45%</td>
<td>36.27%</td>
<td>55.71%</td>
<td>34.38%</td>
</tr>
<tr>
<td>ArrayListMultimap</td>
<td>665</td>
<td>1054</td>
<td>84.21%</td>
<td>82.54%</td>
<td>71.55%</td>
<td>76.26%</td>
<td>62.71%</td>
</tr>
<tr>
<td>LinkedHashMultimap</td>
<td>733</td>
<td>1031</td>
<td>81.58%</td>
<td>81.02%</td>
<td>68.94%</td>
<td>78.96%</td>
<td>64.99%</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>696</td>
<td>986</td>
<td>83.05%</td>
<td>82.79%</td>
<td>73.11%</td>
<td>78.37%</td>
<td>68.66%</td>
</tr>
<tr>
<td>ImmutableListMultiset</td>
<td>1059</td>
<td>1100</td>
<td>62.13%</td>
<td>62.13%</td>
<td>55.66%</td>
<td>61.31%</td>
<td>50.91%</td>
</tr>
<tr>
<td>ConcurrentHashMultiset</td>
<td>675</td>
<td>991</td>
<td>82.50%</td>
<td>83.85%</td>
<td>74.91%</td>
<td>82.54%</td>
<td>68.21%</td>
</tr>
<tr>
<td>.LinkedListMultimap</td>
<td>1091</td>
<td>1100</td>
<td>57.10%</td>
<td>57.10%</td>
<td>46.13%</td>
<td>51.82%</td>
<td>44.62%</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>1004</td>
<td>1053</td>
<td>67.33%</td>
<td>67.13%</td>
<td>64.99%</td>
<td>67.13%</td>
<td>62.87%</td>
</tr>
<tr>
<td>LinkedListHashMultiset</td>
<td>1083</td>
<td>1099</td>
<td>56.59%</td>
<td>54.23%</td>
<td>45.95%</td>
<td>53.78%</td>
<td>41.27%</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>1099</td>
<td>1100</td>
<td>47.45%</td>
<td>46.22%</td>
<td>47.45%</td>
<td>46.64%</td>
<td>46.22%</td>
</tr>
</tbody>
</table>

**Legend**
Columns *Traces, Min, Max* reports the number of traces, from smallest to largest, fed to the subject class.
Columns *Equivalence, Weak, Strong, Event Equivalence and Event Subsumption* identify the inference criteria.
Column *Subject Class* identifies the case studies.
Column *r-swap* identifies the permutation operator.
Columns *Specificity* presents the specificity values obtained with the different inference criteria.
TABLE 7
Comparative evaluation of the specificity of the inference criteria of \(GK\)-tail and \(GK\)-tail+ for the del permutation operator

<table>
<thead>
<tr>
<th>Subject Class</th>
<th>Traces</th>
<th>Specificity</th>
<th>del</th>
<th>Event Equivalence</th>
<th>Event Subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashBiMap</td>
<td>657</td>
<td>1053</td>
<td></td>
<td>84.47%</td>
<td>82.17%</td>
</tr>
<tr>
<td>ArrayListMultimap</td>
<td>171</td>
<td>268</td>
<td>99.42%</td>
<td>99.42%</td>
<td>98.51%</td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>236</td>
<td>282</td>
<td>99.65%</td>
<td>99.65%</td>
<td>99.58%</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>202</td>
<td>268</td>
<td>98.51%</td>
<td>98.51%</td>
<td>98.81%</td>
</tr>
<tr>
<td>ImmutableListMultiset</td>
<td>204</td>
<td>650</td>
<td>99.51%</td>
<td>99.51%</td>
<td>99.51%</td>
</tr>
<tr>
<td>ConcurrentHashMapMultiset</td>
<td>171</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>LinkedListMultimap</td>
<td>171</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>ImmutableBiMap</td>
<td>453</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>LinkedHashSetMultiset</td>
<td>238</td>
<td>550</td>
<td>97.48%</td>
<td>97.48%</td>
<td>97.48%</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>499</td>
<td>1099</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Median (first ten classes)</td>
<td></td>
<td></td>
<td>Median (first ten classes)</td>
<td>97.90%</td>
<td>97.69%</td>
</tr>
<tr>
<td>DateTime</td>
<td>277</td>
<td>499</td>
<td>100.00%</td>
<td>96.39%</td>
<td>95.90%</td>
</tr>
<tr>
<td>DateMidnight</td>
<td>309</td>
<td>395</td>
<td>95.47%</td>
<td>95.47%</td>
<td>94.85%</td>
</tr>
<tr>
<td>Duration</td>
<td>368</td>
<td>499</td>
<td>100.00%</td>
<td>93.75%</td>
<td>100.00%</td>
</tr>
<tr>
<td>HashMultiset</td>
<td>499</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Multigraph</td>
<td>499</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>HashMultimap</td>
<td>499</td>
<td>499</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>TrieMultiset</td>
<td>171</td>
<td>171</td>
<td>99.42%</td>
<td>99.42%</td>
<td>99.42%</td>
</tr>
<tr>
<td>SingleGraph</td>
<td>295</td>
<td>295</td>
<td>99.66%</td>
<td>99.66%</td>
<td>99.66%</td>
</tr>
<tr>
<td>Median (all the classes)</td>
<td></td>
<td></td>
<td>Median (all the classes)</td>
<td>98.53%</td>
<td>98.21%</td>
</tr>
</tbody>
</table>

Legend
Columns \(Traces\), \(Min\), \(Max\) reports the number of traces, from smallest to largest, fed to the subject class.
Columns \(Equivalence\), \(Weak\), \(Strong\), \(Event Equivalence\) and \(Event Subsumption\) identify the inference criteria.
Column \(Subject Class\) identifies the case studies.
Column \(del\) identify the permutation operator.
Columns \(Specificity\) presents the specificity values obtained with the different inference criteria.

Weak Subsumption criteria. Similarly, the specificity of the models generated with the Event Subsumption criterion is consistent with the specificity of the models generated with the Strong Subsumption criterion. The precision of \(GK\)-tail is slightly better than the precision of \(GK\)-tail+ in average, because merging states after inferring the constrains as done in \(GK\)-tail more likely produces a model that may overfit the traces than by merging states before inferring constraints as done in \(GK\)-tail+. The recall and performance results reported in Sections 3.4 and 3.8, respectively, indicate that \(GK\)-tail+ merges many more states than \(GK\)-tail, and thus confirm this observation.
The different specificity of the models generated with the different criteria, which is lower for the models generated with Strong Subsumption and Event Subsumption than for the models generated with Equivalence, Weak Subsumption and Event Equivalence, indicates that the choice of GK-tail or GK-tail+ depends on the relative importance of rejecting as many as possible illegal behaviors over accepting as many as possible legal ones.

3.6 Balanced Classification Rate

Table 8 summarizes the recall and specificity of the criteria of GK-tail+ and Gk-Tail in terms of the balance classification rate (BCR) computed for the subject classes. The values reported in the table confirm that GK-tail and GK-tail+ perform comparably. In particular, the Event Equivalence criterion of GK-tail+ performs similarly to the Equivalence and Weak Subsumption criteria of Gk-Tail, with an average difference of BCR values less than 1% and a maximum difference of 4.97%, while the Event Subsumption criterion GK-tail+ performs similarly to the Strong Subsumption criterion of Gk-Tail, with an average difference of BCR values less than 1% and a maximum difference less than 3.1%.

All the criteria perform better for the top ten than the bottom ten classes. As already observed in Section 3.4, the low BCR values of the bottom ten classes are due to inaccurate sampling of the execution space, as low recall values indicate. In fact, the bottom ten classes are characterized by the availability of a small number of traces compared to the size of the classes.

The similarity of the corresponding criteria confirmed by the BCR values indicates the execution time, which we discuss in Section 3.8, as the key distinguishing feature between the GK-tail+ and Gk-Tail criteria.

3.7 Degree of Dependence from the Number of Traces

In this paper, we performed most of the experiments with 10-fold validation. In this section we study the impact of the number of traces used in the inference process on the quality of the models produced by GK-tail and GK-tail+. Fewer traces may impact positively on the specificity of the inferred models and negatively on the recall, thus, we focus our validation on the recall that we measure for the models inferred with 10-fold, 4-fold and 2-fold cross validation, that is, using 90%, 75% and 50% of the available traces for the inference, respectively. We limit the analysis to the top ten subject classes.

Table 9 reports the recall values for the models inferred with 4-fold and 2-fold cross validation, which can be compared with the recall values for the models inferred with 10-fold cross validation reported in Table 4.

The box plot in Figure 10 compares the values of recall for all the three settings. Results indicate that the recall values gracefully degrade when fewer traces are available. For instance, when halving the number of available traces, the recall decreases only by a small fraction. Results also show that changing the number of traces does not impact on the relative performance of the five criteria: All the criteria perform similarly, with the two GK-tail+ criteria performing slightly better than the others.

3.8 Performance

The experimental data discussed in the previous subsections indicate that the recall and specificity of the equivalence criteria of GK-tail and GK-tail+ are comparable. In particular the Event Equivalence criterion is comparable to the Equivalence and Weak Subsumption criteria, and the Event Subsumption criterion is comparable to the Strong Subsumption criterion.
In this section we investigate the efficiency of these criteria and we show that Event Equivalence and Event Subsumption criteria can be computed definitely faster than the other criteria, thus improving the scalability of GK-tail+ over GK-Tail.

We measure the costs of inferring the models with all the criteria when applied to the subject programs listed in Table 2. Table 10 reports both the cost of the individual steps and the total inference time for all the criteria.

The Merging Trace step is described in Section 2.2, and is the same for all the considered criteria of both GK-tail+ and GK-tail.

The Merging States step includes both the generation of the initial FSM and the merging of the states. GK-tail+ creates the initial FSM and merges the states referring only to the events, as described in Sections 2.3 and 2.4, while GK-tail refers both to events and values to identify the states to be merged, as discussed in [6].

Consequently, the Generating Constraints step is executed at different moments in the GK-tail and the GK-tail+ in-
Fig. 10. Aggregated data of recall for 10-fold, 4-fold and 2-fold cross-validation

Table 10
Comparative evaluation of the execution time of the inference criteria of\textit{GK-tail} and \textit{GK-tail+}

<table>
<thead>
<tr>
<th>Inference Step</th>
<th>\textit{GK-tail}</th>
<th>\textit{GK-tail+}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Equivalence</td>
<td>Weak</td>
</tr>
<tr>
<td>Merging Traces</td>
<td>201 sec</td>
<td>2549 sec</td>
</tr>
<tr>
<td>Merging States</td>
<td>2549 sec</td>
<td>1572 sec</td>
</tr>
<tr>
<td>Generating Constraints</td>
<td>4794 sec</td>
<td>953 sec</td>
</tr>
<tr>
<td>Inference Time</td>
<td>7545 sec</td>
<td>6567 sec</td>
</tr>
</tbody>
</table>

Legend: The table reports the median value of the time required to complete each step of the inference process and the overall inference process for each criterion.

Fig. 11. Visual comparative evaluation of the execution time of the inference criteria of \textit{GK-tail} and \textit{GK-tail+}

Inference processes. \textit{GK-tail} generates constraints from the annotations associated with the events, while \textit{GK-tail+} generates constraints from the annotations associated with the transitions in the final gFSM, as described in Section 2.5. Since \textit{GK-tail} generates the constraints before merging the states, the cost of this step is the same for all the \textit{GK-tail} criteria. On the contrary, \textit{GK-tail+} executes this step after the state merging process, thus the cost of this step differs for the Event Equivalence and Event Subsumption criteria.

Figure 11 visually summarizes the inference time of the criteria reported in Table 10, detailing the cost of the individual steps. Both the \textit{GK-tail+ Event Equivalence} and \textit{Event Subsumption} criteria perform much better the corresponding \textit{GK-tail} criteria. The \textit{GK-tail+ Event Equivalence} criterion requires half of the time than the \textit{GK-tail Equivalence} and \textit{Weak Subsumption} criteria, and the \textit{GK-tail+ Event Subsumption} criterion is four time faster than the \textit{GK-tail Strong Subsumption} criterion. This impressive performance improvement derives from the new \textit{GK-tail+} algorithmic organization that generates the constraints at the end of the inference process and simplifies the state merging process. By generating constraints at the end of the inference process, \textit{GK-tail+} reduces the amount of expensive invocations of the inference engine. In fact, \textit{GK-tail+} invokes the inference engine for each transition in the final gFSM, while \textit{GK-tail} invokes the engine for each event in the merged traces, and this might reduce the amount of invocations of the inference engine by an order of magnitude. The simplified \textit{GK-tail+} state merging process does not require comparing constraints as in \textit{GK-tail}, and thus eliminates another expen-
sive activity.

Figure 12 shows the comparative evaluation of the different phases of the \(\text{GK-tail}^+\) and \(\text{GK-tail}\) criteria. The box plot of the merging traces step confirms the marginal contribution of this step, which is shared among all the criteria, to the overall inference cost. The box plots of the merging states and generating constraints steps indicate the major contribution of these two activities to the performance improvements of \(\text{GK-tail}^+\) over \(\text{GK-tail}\). While the merging state step presents some variability in the runtime cost, the generating constraints step is consistently less expensive for the \(\text{GK-tail}^+\) criteria with respect to the corresponding \(\text{GK-tail}\) criteria. The box plots of the total inference time confirm the substantial performance improvement of \(\text{GK-tail}^+\) over \(\text{GK-tail}\).

Figure 13 and 14 show the variation of the runtime cost of the inference process with respect to the number of input events.

Figure 13 presents the scatter plot of the inference time for the \(\text{Equivalence}, \text{Weak Subsumption}\), and \(\text{Event Equivalence}\) criteria with respect to the number of events in the input traces, that is with respect to the sum of the length of all the input traces. Figure 14 presents the scatter plot of the inference time for the \(\text{Strong Subsumption}\) and \(\text{Event Subsumption}\) criteria. In both figures, the small black arrow at the top indicates the existence of outlier values in correspondence of the arrow.

The scatter plots confirm that the inference time of the \(\text{Event Equivalence}\) and \(\text{Event Subsumption}\) criteria is significantly and systematically lower than the inference time of the other criteria, and indicate that the inference time grows gracefully when the number of traces increases, suggesting a good scalability of the approach, in particular for the \(\text{Event Equivalence}\) criterion.

To confirm the hypothesis that the improvement of \(\text{GK-tail}^+\) over \(\text{Gk-Tail}\) depends on both the reduced amount of invocations of Daikon and the increased amount of samples for each Daikon invocations, we compare the Daikon invocations when executing the \(\text{GK-tail}^+\) and \(\text{Gk-Tail}\) criteria.

Table 11 reports the number of calls to the Daikon inference engine when analysing the first 10 subject classes with the \(\text{GK-tail}^+\) and \(\text{Gk-Tail}\) criteria. \(\text{Gk-Tail}\) executed with the three criteria (\(\text{Equivalence}, \text{Weak and Strong}\)) interacts with Daikon in the same way, and thus perform the same number of calls that is reported once for all the criteria in the figure. \(\text{GK-tail}^+\) dramatically reduces the number of calls to Daikon. The median number of calls to Daikon with \(\text{Gk-Tail}\) is 1338 for all the criteria, while with \(\text{GK-tail}^+\) the median is 320 and 222, with the \(\text{Event Equivalence}\) and the \(\text{Event Subsumption}\) criteria, respectively, with a reduction of number of calls between 76% and 83% with respect to \(\text{Gk-Tail}\).

The reduced number of calls to Daikon comes with an increased amount of samples for each call, which goes from a median number of 1.72 samples for \(\text{Gk-Tail}\) to a median number of 6.77 and 7.12 samples for \(\text{GK-tail}^+\) with the \(\text{Event Equivalence}\) and the \(\text{Event Subsumption}\) criteria, respectively, that corresponds to a 3.9X and 4.1X improvement, respectively. The box plot in Figure 15 provides an intuitive visualisation of the increasing amount of samples for each Daikon invocation when moving from the \(\text{Gk-Tail}\) to the \(\text{GK-tail}^+\) criteria, which can have only a positive impact on the precision of Daikon results.

Overall, the empirical results indicate the relevant improvement of \(\text{GK-tail}^+\) over \(\text{Gk-Tail}\): The \(\text{Event Equivalence}\) and \(\text{Event Subsumption}\) criteria defined in \(\text{GK-tail}^+\) largely improve in performance and scalability over the corresponding \(\text{Gk-Tail}\) criteria, with comparable recall and specificity.

3.9 Threats to Validity

The main threats to the validity of the experimental results reported in this paper derive from the choice of values for the parameter \(k\) and the generation of illegal traces used in the experiments.
Fig. 13. Inference time of Equivalence (GK-tail), Weak Subsumption (GK-tail) and Event Equivalence (GK-tail+) with respect to the number of input events

Fig. 14. Inference time of Strong Subsumption (GK-tail) and Event Subsumption (GK-tail+) with respect to the number of input events

Different values of $k$ produce different models and hence different results. Section 3.3 reports the results of our experimental investigation on the impact of different values of $k$ that confirm that $k = 2$ is an excellent choice and that small changes to the values of $k$, for instance $k = 1$ or $k = 3$, do not significantly affect the experimental results.

We generated the (likely) illegal traces by changing the order of events in the monitored traces, with the risk of generating legal traces. We mitigate such risk by excluding the mutated traces that match a trace known to be legal. Although this does not completely eliminate the risk of incidentally generating legal traces, it contributes to improve the quality of the experiments. Due to the large number of generated traces and the complexity of the manual checking,
TABLE 11
Comparative evaluation about the use of Daikon between GK-tail+ and GK-tail

<table>
<thead>
<tr>
<th>Subject Class</th>
<th>Number of Calls</th>
<th>Number of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GK-tail</td>
<td>GK-tail+</td>
</tr>
<tr>
<td></td>
<td>Event</td>
<td>Event</td>
</tr>
<tr>
<td></td>
<td>Equivalence</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td>HashBiMap</td>
<td>1478</td>
<td>361</td>
</tr>
<tr>
<td>ArrayListMultimap</td>
<td>1772</td>
<td>420</td>
</tr>
<tr>
<td>LinkedHashMultimap</td>
<td>1756</td>
<td>396</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>2739</td>
<td>499</td>
</tr>
<tr>
<td>ImmutableListMultiset</td>
<td>259</td>
<td>125</td>
</tr>
<tr>
<td>ConcurrentHashMultiset</td>
<td>1105</td>
<td>302</td>
</tr>
<tr>
<td>LinkedListMultimap</td>
<td>1846</td>
<td>339</td>
</tr>
<tr>
<td>ImmutableBiMap</td>
<td>114</td>
<td>47</td>
</tr>
<tr>
<td>LinkedHashMultiset</td>
<td>1199</td>
<td>228</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>153</td>
<td>21</td>
</tr>
<tr>
<td>Median</td>
<td>1338</td>
<td>320</td>
</tr>
</tbody>
</table>

Legend

Columns Subject Class identifies the case studies.
Columns Equivalence, Weak, Strong, Event Equivalence and Event Subsumption identify the inference criteria of GK-tail and GK-tail+.
Columns Number of Calls presents the number of times Daikon has been executed.
Columns Number of Samples presents the number of samples Daikon is executed on.

Fig. 15. Number of samples processed by Daikon with the GK-Tail and and GK-tail+ criteria

we could not verify all the traces manually. We inspected a set of sample traces for each subject class, and we did not find any legal trace, thus confirming the validity of the generated illegal traces.

4 RELATED WORK

In this section, we frame the contribution of this paper in the context of inference techniques that dynamically analyze execution traces. We discuss approaches for learning simple FSMs, for generating augmented FSMs and for inferring other kinds of models of relations among events.

4.1 Learning FSMs

Learning FSMs from execution traces is an instance of the well-known regular inference problem, which consists of identifying a language from a set of sample sentences. In the early seventies, Biermann and Feldman proposed the seminal and inspiring kTail algorithm, a notable example of
procedural trace-based inference, which refines an initial Prefix Tree that combines the input traces into a FSM with an iterative state merging process [31]. GK-tail+ extends kTail to gFSMs.

Several variants of state-based inference address different contexts and with various goals: Cook and Wolf’s approach reduces the size of the model inferred with kTail [33]; Ammons et al.’s technique extends kTail to generate a probabilistic FSM [44]; Walkinshaw et al.’s method extends the state-based inference process with the capability to deal with a set of user-provided temporal rules that should not be violated by the language accepted by the inferred model [20]. Walkinshaw et al.’s rules steer the inference process, and improve the accuracy of the final model. Lo et al.’s rule inference process fully automates Walkinshaw et al.’s approach [34].

Some inference approaches take advantage of pre- and post-conditions that may be present in the code. The approach proposed by de Caso et al. requires compatible pre-conditions, respectively post-conditions, for the operations that can lead to, respectively can leave from, a given state [45]. Although effective, these strategies are inevitably limited to software operations documented with pre- and post-conditions.

The kBehavior approach proposed by Mariani et al. implements an alternative inference strategy that exploits similarities between sequences of events rather than similarity between states to infer FSMS [14], [46]. Algorithms that exploit state merging process and algorithms based on similarities between sequences of events are characterized by complementary precision and recall [30].

An alternative strategy is the one defined both in the Synoptic approach proposed by Beschastnikh et al. [4] and the approach proposed by Lo et al. [34], which exploits mined temporal rules to build the final FSM.

Declarative inference algorithms, like InvariMint [17], [47], define the inference algorithm in terms of a set of properties that the final model must satisfy, without worrying about the inference process. For instance, a property may specify that two events that consistently occur together in all the traces should necessarily occur together also in the final model. This learning style increases the control over the characteristics of the resulting model compared to procedural algorithms, but requires to identify a-priori the relevant properties that must be satisfied by the inferred model, which might be hard for complex software systems and non-trivial application domains.

Approaches that generate FSMS from execution traces have been combined with testing and monitoring techniques to obtain additional traces and improve the accuracy of the inferred models: Dallmeier et al.’s approach systematically generates test cases that cover the possible sequences of operations to fully discover software protocols [7]; Bertolino et al.’s technique uses the results of testing and monitoring to improve the inference process in the context of service-oriented applications [48]; The TTT algorithm proposed by Isberner et al. improves the accuracy of the inferred model in the presence of long traces [49].

The many approaches that infer simple FSMS provide a solid background for the approaches to generate gFSMs that we discuss in the next subsection and that include GK-tail+.

4.2 Learning Augmented FSMS

Approaches that infer FSMS augmented with various kinds of information capture a wider set of details of the monitored behavior.

The most relevant approaches to learn augmented FSMS are the state-based inference algorithms, which process traces that include information about both the sequences of the monitored events and the sequences of concrete states that have been traversed between the execution of the events. State-based inference algorithms rely on a state abstraction function that computes the abstract states in the FSMS from the concrete states in the traces, annotates the abstract states with the state invariants computed with the abstraction function, and infers the transitions between states from the sequence of monitored events. Well-known instances of state-based inference algorithms are ADABU, which works with Java applications [29]; ReAjax, which works with Web applications [28]; and Revolution, which provides an incremental version of the state-based inference process [27].

The SEKT algorithm presented in [26] increases the level of automation of state-based inference by mining constraints that characterize the states in the model. Although both SEKT and GK-tail+ use Daikon to infer the constraints, they use the inferred constraints for different purposes. SEKT exploits the inferred constraints to identify the states in the model, while GK-tail+ exploits the inferred constraints to augment transitions with guard conditions.

Krika et al. have recently demonstrated that state-based inference algorithms might provide more accurate information than trace-based inference algorithms [26], but suffer from the limitation of requiring the logging of the state of the application in addition to the events, which might be often infeasible or simply too expensive.

Only few techniques address the challenging task of inferring FSMS augmented with information about the values of the parameters associated with the monitored events. The KLFA approach of Mariani and Pastore generates FSMS with labels associated with transitions that encode both event names and information about the recurrence of the parameter values across events [50]. The KLFA approach captures a different kind of information than the GK-tail+ guards. For example, KLFA may infer that a set of login and purchase events have been executed by a same user, but cannot learn constraints over the parameters, for instance the information that the username is longer than N characters or that the user has purchased more than M items, which are conditions that GK-tail+ can capture and encode in the guarded FSMS. Lo et al. show that models generated by KLFA are usually less accurate than models generated by GK-Tail, when applied to traces that encode sequences of method calls [30].

An interesting body of work has proposed active learning techniques to infer FSMS augmented with guards [21]–[24]. These techniques iteratively generate and execute test cases to produce new traces for the learner, until proving the compliance of the model with the application. Compliance checking is extremely expensive, and is usually performed using either model checking or testing. Model checking can be applied only when the source code is available and is known to suffer when the complexity of the code grows.
Testing can be applied without source code, but compliance checking through testing is always inaccurate and its cost grows with the number of tests to be executed. Differently from active learning approaches, GK-tail+ implements a black-box passive learning strategy, which is always applicable and does not require expensive compliance checking.

The recent Walkinshaw et al.’s MINT algorithm infers FSMs with transitions annotated with classifiers inferred from traces. MINT generates the classifiers with (potentially any) data mining algorithm, and constrains the values that can be assigned to the parameters of a given label, for instance a given method [25].

MINT exploits a learning style complementary to GK-tail+. MINT exploits the inferred constraints during the generalization process, to determine if the states in the model have to be merged, and determines the constraints on a per label basis. GK-tail+ infers the constraints on a per transition basis, that is the same label can be associated with completely different constraints when occurring on different transitions. Thus, MINT is more suitable to infer FSMs where parameters have a significant influence on the sequences of events that follow in the execution, while GK-tail+ is more suitable to infer FSMs where the guards on the transitions constrain the parameter values, but have little influence on the events that follow in the execution.

4.3 Learning Models of Relations Among Events

Some approaches encode the relations among events with other kinds of models, notably event patterns [51], [52] and temporal logic rules [53]–[55]. Event patterns and temporal rules can capture well some temporal logic rules [53]–[55]. Event patterns and temporal rules can capture well some partial but relevant facts about the behavior of complex software systems that can be hardly addressed with techniques that infer FMSs, while FSMs can represent well the full behavior of a software component of medium complexity. In this paper, we presented a technique that can efficiently produce models that capture information about both the sequence of events and the values of the parameters of the events.

5 Conclusions

In this paper, we present a technique to efficiently infer models of the behavior of software systems in the form of guarded finite state machines, which capture the sequences of method calls together with the constraints on the parameter values.

Learning models from program execution traces provides important information about the software behavior without requiring expensive human effort. Useful models shall comprehensively represent the system behavior, limit the amount of illegal behaviors that may be erroneously accepted, and be inferred within a reasonable time budget to scale to cases of interesting size. Dynamically generated models of the software behavior find interesting applications in the fields of specification mining, program comprehension, test case generation, fault diagnosis and bug fixing.

In this paper we address the problem of efficiently learning state-based models augmented with data information.

In our early work, we have investigated the problem of generating guarded finite state machines from execution traces and we proposed GK-tail, an approach that generates accurate guarded finite state machines, that is, models with a high rate of acceptance of valid traces and of rejection of invalid ones, albeit with generation costs that limit the applicability of GK-tail to cases of small size.

In this paper we redefine the inference criteria that characterize GK-tail, and we propose GK-tail+, an approach that embeds the new inference criteria, and generates guarded finite state machines with comparable rates of both accepted valid traces and rejected invalid ones, and with largely improved performance over GK-tail.

In detail this paper presents a new algorithm, GK-tail+, and two new criteria for generating behavioral models that integrate event sequences and parameter values, provides a complete formalization of GK-tail+, and reports a set of experimental results that confirm the comparable effectiveness and the improvement in the efficiency. The experimental results reported in this paper indicate that the difference in the recall and specificity of the models inferred with GK-tail+ and GK-tail is negligible, while the inference time of GK-tail+ is from 50% to 75% lower than the one of GK-tail, depending on the inference criteria. The experiments also indicate a limited growth rate of the inference time with respect to the input events, thus confirming the scalability of the approach.

Acknowledgment

This research has been partially supported by the H2020 Learn project, which has been funded under the ERC Consolidator Grant 2014 program (ERC Grant Agreement n. 646867), and by the Swiss National Foundation (SNF Grant 200021_162409: ASysT: Automatic System Testing).

References


