TkT: Automatic Inference of Timed and Extended Pushdown Automata

Fabrizio Pastore, *Member, IEEE,* Daniela Micucci, *Member, IEEE,* Michell Guzman, and Leonardo Mariani, *Senior Member, IEEE*

Abstract—To mitigate the cost of manually producing and maintaining models capturing software specifications, *specification mining* techniques can be exploited to automatically derive up-to-date models that faithfully represent the behavior of software systems. So far, specification mining solutions focused on extracting information about the functional behavior of the system, especially in the form of models that represent the ordering of the operations. Well-known examples are finite state models capturing the usage protocol of software interfaces and temporal rules specifying relations among system events.
Although the functional behavior of a software system is a primary aspect of concern, there are several other non-functional characteristics that must be typically addressed jointly with the functional behavior of a software system. Efficiency is one of the most relevant characteristics. Indeed, an application that delivers the right functionalities with an inefficient implementation may fail to satisfy the expectations of its users.
Interestingly, the *timing behavior* is strongly dependent on the functional behavior of a software system. For instance, the timing of an operation depends on the functional complexity and size of the computation that is performed. Consequently, models that combine the functional and timing behaviors, as well as their dependencies, are extremely important to precisely reason on the behavior of software systems.
In this paper, we address the challenge of generating models that capture both the functional and timing behavior of a software system from execution traces. The result is the *Timed k-Tail (TkT) specification mining technique*, which can mine finite state models that capture such an interplay: the functional behavior is represented by the possible order of the events accepted by the transitions, while the timing behavior is represented through clocks and clock constraints of different nature associated with transitions.
Our empirical evaluation with several libraries and applications shows that *TkT* can generate accurate models, capable of supporting the identification of timing anomalies due to overloaded environment and performance faults. Furthermore, our study shows that *TkT* outperforms state-of-the-art techniques in terms of scalability and accuracy of the mined models.

Index Terms—Specification mining, dynamic analysis, trace analysis, model inference, timed automata, performance analysis.

1 Introduction

Behavioral models provide an abstract representation of the behavior of a software system, considering for instance various perspectives and different granularity levels. They are the starting point for many development activities, including requirements engineering [1], testing [2], and software evolution [3].

Unfortunately, sound and up-to-date models are seldom available, and in many practical cases the only trustable source of information about the behavior of a system is the system itself. In these cases, specification mining techniques can be used to derive models either dynamically, from the analysis of execution traces, or statically, from the analysis of the source code. Although the mined models inevitably reflect the behavior of the actual system rather than the intended behavior of the system, they can be extremely useful in supporting many activities, including program understanding [4], software evolution [5], verification [6], testing [7], [8], and failure analysis [9], [10], [11], [12], [13].

So far, specification mining focused on the *functional* behavior of software systems, deriving models such as finite state models [14], [15], [16], extended finite state machines [17], [18], [19], temporal rules [20], [21], and program invariants [22], [23].

While the inference of models about the functional behavior of software systems has been extensively investigated, little attention has been paid to the inference of models that capture the relationship between the *functional behavior* of a software system and its *timing behavior*. This relationship is extremely important to establish the correctness of software executions: an execution delivering the right result might be wrong when considering the time spent in the computation, and vice versa a quick execution might be wrong when looking at the delivered result.

Early approaches trying to derive models that combine the functional and timing behaviors considered the inference of timed automata [24], [25], [26], [27], [28], [29]. Unfortunately, these techniques derive one clock timed automata [30], which are effective to capture the delay between consecutive events (e.g., signals processed by control systems [24], [26]), but cannot be used to represent the duration of nested operations, which are common in many software systems.

Instead of using timed automata, Perfume [10] derives fi-

- *D. Micucci, M. Guzman, and L. Mariani are with the Department of Informatics, Systems, and COmmunications, University of Milano - Bicocca, 20126, Milano, Italy.*
  Email: {daniela.micucci, michell.guzman, leonardo.mariani}@unimib.it.
- *F. Pastore is currently with SnT Centre - University of Luxembourg, L1855, Luxembourg, Luxembourg. Most of his work was done under the affiliation of University of Milano - Bicocca.*
  Email: {fabrizio.pastore}@uni.lu.

Manuscript received Month Day, YEAR; revised Month Day, YEAR.

nite state models annotated with information about resource consumption, including time consumption. Although Perfume is primarily meant to produce models that can simplify the identification of suspicious behaviors, the generated models also capture the relation between the timing and functional behavior.

This paper presents the *Timed k-Tail* (*TkT*) specification mining technique, which addresses the challenge of deriving accurate models that coherently capture the functional and timing aspects of a software system. In particular, *TkT* can derive both *timed automata* [31] and *extended pushdown timed automata* [32] from execution traces. *TkT* exploits the natural structure of the execution of a program, which consists of a possibly nested sequence of operations, to automatically derive information about the duration of the operations, as well as their dependencies.

This paper extends our previous work on the inference of timed models [33] in several ways: (i) the inference algorithm has been extended with the capability to derive extended pushdown timed automata that better capture the structure of a software execution in presence of iterative behaviors (e.g., loops), (ii) the technique is rigorously and formally presented, using algorithms, examples, and formal definitions, (iii) the effectiveness of the two models that can be derived with *TkT* has been empirically compared, and (iv) the empirical evaluation has been extended to a larger set of systems and trace files considering seven additional program versions from the Apache Commons Collections [34] and Apache Commons Math [35] libraries.

In a nutshell, the major contributions of this paper are:

- The definition of *TkT*, a specification mining technique for the generation of timed automata and extended pushdown automata.
- The implementation of a publicly available version of *TkT*, downloadable from https://github.com/lta-disco-unimib-it/tkt.
- An empirical evaluation that both investigates the effectiveness of the various configurations of *TkT*, including the two classes of timed models that can be generated, and compares *TkT* to Perfume.

The paper is organized as follows. Section 2 provides background information. Section 3 presents the *TkT* specification mining technique using a simple running example. Section 4 presents our empirical evaluation. Section 5 discusses related work. Section 6 provides final remarks.

2 Traces and Timed Models

Given a set of traces collected from a running system, Timed k-Tail (*TkT*) can automatically generate either a Timed Automaton (TA) [31] or an Extended Pushdown Timed Automaton (EPTA) [32] that generalizes the observations in the traces. The generated model represents the behavior of the monitored system in terms of both the sequences of operations that the system can execute and the timing of these operations.

A TA measures the time taken by an operation including the time taken by its sub-operations. For instance, if an operation that processes an order in an online shopping system includes two finer-grained operations, one that executes a bank transaction and one that sends a confirmation email, a TA represents in the model the overall duration of the order processing, including the execution of both the transaction and the email dispatching.

An EPTA measures the time taken by an operation excluding the time taken by its sub-operations. For instance, in the example of the shopping system's order, an EPTA represents in the model the time taken to process an order excluding the time taken for the bank transaction and the email dispatching. This is achieved by properly pushing and pulling clocks to and from a stack while entering and exiting methods.

TkT generates Timed Automata (TAs) and Extended Pushdown Timed Automaton (EPTAs) by augmenting the k-Tail algorithm [36], which can only produce simple automata, with the ability to generate clocks, reset operations, clock constraints, and operations to store/restore clock values using a valuation stack.

In the rest of this section, we formally define the kind of traces processed by *TkT* (Section 2.1), the models that *TkT* can generate (Section 2.2), and we provide background information about the k-Tail technique (Section 2.3).

2.1 Timed Execution Traces

Definition 1 (Timed execution trace). A *timed execution trace* of length n is defined as a temporally ordered sequence $event_1 \dots event_n$, where $event_i = (type_i, op_i, time_i)$, with

- $type \in \{B, E\}$, where B and E indicate whether the operation begins or ends,
- *op* is a label that identifies the operation,
- $time \in \mathbb{N}_0^+$ is the timestamp of the event.

Given an event e, we denote with $type(e)$, $op(e)$, and $time(e)$ the corresponding element of e.

Definition 2 (Well-formed timed trace). A well-formed timed trace satisfies the following properties:

- *time does not decrease*: $time_i \leq time_j \; \forall i \leq j$
- *events are paired*: when an operation *op* starts, it produces the event $(B, op_i, time_i)$ and when it ends, it produces the event $(E, op_j, time_j)$ with $time_i \leq time_j$. Given an event e of type either B or E referring to an operation *op*, we indicate the event of the other type referring to the same operation *op* with $pair(e)$.
- *nesting of the operations is satisfied*: all operations started after the beginning of an operation *op* must end before the operation *op* ends.

An *execution trace* is a timed execution trace without time information associated with the events.

2.2 Timed Models

2.2.1 Timed Automata

In this paragraph we extend the definition of Timed Automata provided by Alur et al. in [31] to explicitly characterize the mapping between software operations reported in execution traces and transitions of the timed automaton.

Definition 3 (Timed Automata). A *Timed Automaton* is a tuple (Σ, S, s_0, C, TR), where

- Σ is a finite input alphabet;
- S is a finite set of states;
- $s_0 \in S$ is the initial state;
- C is a finite set of clocks, initially set to $\perp$ (i.e., initially undefined);
- $TR \subseteq (S \times S \times \Sigma \times \{B, E\} \times 2^C \times G(C))$ is a set of transitions. A transition $tr = (s_a, s_b, \sigma, t, g, r) \in TR$ from a state s_a to a state s_b takes place on an input symbol $\sigma \in \Sigma$. The value t indicates that the operation either begins (symbol B) or ends (symbol E). Further, g is a guard condition defined on the set of clocks C that must evaluate to true to enable the firing of the transition. Guard conditions are defined as boolean propositions that join predicates by using the logical operators $\wedge$ (which indicates conjunction). Predicates involving undefined clocks evaluate to true. Finally, r is a set of clocks reset to 0 when the transition is fired.

Given a timed trace $tt = event_1 \dots event_n$, with $event_i = (type_i, op_i, time_i)$ and a timed automaton $TA = (\Sigma, S, s_0, C, TR)$, the TA accepts tt if and only if $\exists tr_1 \dots tr_n$, with $tr_i = (s_{i-1}, s_i, e_i, t_i, g_i, r_i) \in TR$ $\forall i = 1 \dots n$ that satisfies the following conditions: $e_i \in \Sigma$, $e_i = op_i$, $t_i = type_i$, and each constraint in g_i evaluates to true, according to the values of the clocks. Note that the value of a clock is determined by the time passed between the last reset operation executed on that clock and the timestamp of the current event in the trace. In the context of *TkT*, clocks are opportunistically reset when operations start (on begin transitions) and clocks are checked against guards when the operations end (on end transitions). Guards on clocks that are undefined because they have never been reset evaluate to true.

Note that the TA formalism considered in this article differs from the Input/Output Timed Automata formalism made popular by the UPPAAL toolset [37] since we do not include state invariants in the model. Moreover, we rely on the original terminology adopted by Alur et al., that is, we use the term *state* instead of *location* to indicate the states of the automaton.

2.2.2 Extended Pushdown Timed Automata

Extended Pushdown Timed Automata (EPTAs) have been introduced in [32] to enable the modelling of procedural calls based on the Pushdown Timed Automata formalism [38]. More formally,

Definition 4 (Extended Pushdown Timed Automata). An *Extended Pushdown Timed Automata* is a tuple $(\Sigma, S, s_0, C, \Gamma, TR)$, where

- Σ is a finite input alphabet;
- S is a finite set of states;
- $s_0 \in S$ is the initial state;
- C is a finite set of clocks, initially set to $\perp$ (i.e., initially undefined);
- Γ is valuation stack, initially empty;
- $TR \subseteq (S \times S \times \Sigma \times \{B, E\} \times 2^C \times G(C) \times \{\epsilon, Store, Restore\})$ is the set of transitions $tr = (s_a, s_b, \sigma, t, g, r, sop)$. TR follows the definition used for TAs except that, in the case of EPTAs, a transition taking place on an input symbol σ may imply the execution of a stack operation (*sop* in the formula). The stack operation *Store* pushes the values of all clocks belonging to C on the valuation stack, while the stack operation *Restore* pops the last set of clock values added to the valuation stack and assign them to the corresponding clock variables. If the reset set r is not null, the clocks in r are reset after performing the stack operation *Store/Restore*.

We use EPTAs to capture the time spent within single operations (i.e., procedure calls) without considering the time spent in nested operations, if any. This is achieved thanks to the valuation stack, which is used to store the value of the clocks when a nested operation is under execution and to restore it when the nested operation terminates.

Similarly to the case of the *Timed Automaton* described above, given a timed trace $tt = event_1 \dots event_n$, with $event_i = (type_i, op_i, time_i)$ and an $EPTA = (S, s_0, C, E, TR)$, the EPTA accepts tt if and only if $\exists tr_1 \dots tr_n$, with $tr_i = (s_{i-1}, s_i, e_i, t_i, g_i, r_i) \in TR$ $\forall i = 1 \dots n$ that satisfies the following conditions: $e_i = op_i$, $t_i = type_i$, and each constraint in g_i evaluates to true, according to the values of the clocks.

In this paper, we show how inferred TAs and EPTAs can be used to verify well-formed timed traces. This is a typical application scenario for specification mining techniques: the models are first derived from traces collected during valid executions and then used to reveal anomalous behaviors in a new set of traces [9], [11], [12]. However, inferred models can also be used in alternative contexts. For example, they can be compared to reveal the presence of conflicts in concurrent software changes [39]. For this reason, it is important to remark that the TA and EPTA inference process algorithms proposed in this paper do not formally guarantee that only well-formed traces are accepted, and thus the inferred model may end up accepting traces that are not well-formed.

2.3 k-Tail

k-Tail is an inference algorithm that can generate a regular finite state automaton from a set of execution traces [36].

It works in two steps. In the first step, it generates a Prefix Tree Automaton (PTA), that is, a tree-like automaton where each branch accepts a different execution trace. Traces with common prefixes share sub-branches in the model. This representation is similar to the one used by *TkT* to produce an initial automaton that accepts the input timed execution traces (see Section 3.2). Note, however, that k-Tail ignores time information.

In the second step, k-Tail iteratively modifies the automaton by merging the states of the model that are likely to represent a same state of the program. The state merging process produces a more compact and more general model than the initial model, in fact the resulting automaton has

fewer states and accepts more execution traces than the initial one.

The states in the model that likely represent the same states of the program are heuristically identified based on their k-future, which consists of the set of execution traces of maximum length k that can be accepted by that state. Two states with the same k-future are assumed to be equivalent and thus are merged. The concept of equivalent states is extended to cope with time information and exploited also by *TkT*, as described in Section 3.3.

3 TIMED K-TAIL

Figures 1 and 2 give a visual overview of the outputs of the five steps of the *TkT* algorithm when it is used to generate TA and EPTA from the same execution traces. The figures are used throughout this section to describe the different steps of the algorithm.

The input to *TkT* is a set of timed traces with information about the ordering and the timing of the operations executed by a program, see for instance *Trace 1* and *Trace 2* in Figure 1. For each executed operation, a trace includes two events that mark the beginning and the end of the operation, respectively. We graphically indicate the beginning and the end of a same operation with the letters *B* and *E* respectively, plus a thick line that connects them. For instance, the first two events in *Trace 1* represent the beginning and the end of the `ProcessWebOrder` operation. Finally, each event has a timestamp indicating when the event has been observed.

In the first step, *Trace Normalization*, *TkT* normalizes the timing information in the traces, which might have been collected at different times, by assigning time 0 to the first event in each trace and adjusting the times associated with the other events in the traces consistently to preserve time difference between events.

In the second step, *Automaton Initialization*, *TkT* produces an initial automaton where each trace corresponds to a branch in the model, see for instance the *Initial Automaton* in Figure 1. The automaton is annotated with *relative clocks* that measure the time taken by each operation logged in the trace to complete. In particular, each clock is reset when an operation starts and checked with an equality constraint when the operation completes. Depending on the kind of model that is generated, the equality constraint may or may not consider the time spent in nested operations (the TA considers the time spent in nested operations, the EPTA does not consider it). *TkT* can also be configured to add a clock measuring absolute time to the automata.

In the third step, *State Merging*, *TkT* merges equivalent states, which are states in the model that are likely to represent a same state of the monitored system. Merging states might lead to multiple transitions, with the same label, that start and end in the same states, that is, redundant transitions that represent the same event (same event name and same event type). In such a case, *TkT* merges these transitions into a single transition with the same label and merges annotations (i.e., the resulting transition includes all resets and clock constraints in the merged transitions). Thus, the more states are merged, the more reset operations and equality constraints are accumulated on the same transitions of the model. After the state merging process has completed, the resulting model represents the behavior of the monitored software quite extensively, see for instance the *Merged Automaton* in Figure 1. Note that while the information about the sequences of operations that can be performed is a generalization of the sequences reported in the traces, the timing information is still encoded as a simple set of equality constraints that match the observations reported in the traces. The next two steps of the process further elaborate the timing information to produce more general and flexible timing constraints.

In the fourth step, *Clock Refinement*, *TkT* identifies redundant clocks, that is, distinct clocks that measure the duration of exactly the same operation, and transforms them into individual clocks that are reset once and checked multiple times. This transformation reduces the number of clocks in the model and increases the number of equality constraints on a same clock associated with a same transition. See for instance the *Automaton with Refined Clocks* in Figure 1.

In the last step, *Guards Generation*, *TkT* processes the information on each transition to transform the many equality checks on the same clocks into a time interval constraint. *TkT* combines with conjunctions the time interval constraints that refer to different clocks but are associated with a same transition, if any. The resulting model is either a timed automaton or an extended pushdown timed automaton that captures not only the ordering of the events, but also the expected duration of the operations. See for instance the *Timed Automaton* in Figure 1 and the *Extended Pushdown Timed Automaton* in Figure 2.

In the following, we rigorously define each step of the algorithm.

3.1 Trace Normalization

TkT processes timed traces, that is, traces that log the execution of the operations and their durations. Each trace starts with its own timestamp. This is not an issue for relative clocks, but when the inferred timed automaton includes constraints on the absolute time, it is important to make sure all traces refer to the same starting time, otherwise the timing information would not be comparable. We conventionally use 0 as starting time of each trace, thus the normalization process simply subtracts the value of the timestamp of the first event in the trace to the timestamp of each event. For instance, the time associated with the first two events of *Trace 1* is changed from 98483940 and 98483943 to 0 and 3, respectively. This operation is important to align time across multiple traces.

3.2 Automaton Initialization

In this section we describe how *TkT* deals with the initialization of the automaton by presenting first the case of the *Initial Automaton* created in order to generate a *Timed Automaton* and then the *Initial Pushdown Automaton* created to finally build an *Extended Pushdown Automaton*.

3.2.1 Initial Timed Automaton

The *Automaton Initialization* step generates an *Initial Automaton* that accepts all and only the executions stored in the traces (see Figure 1). The *Initial Automaton* is obtained

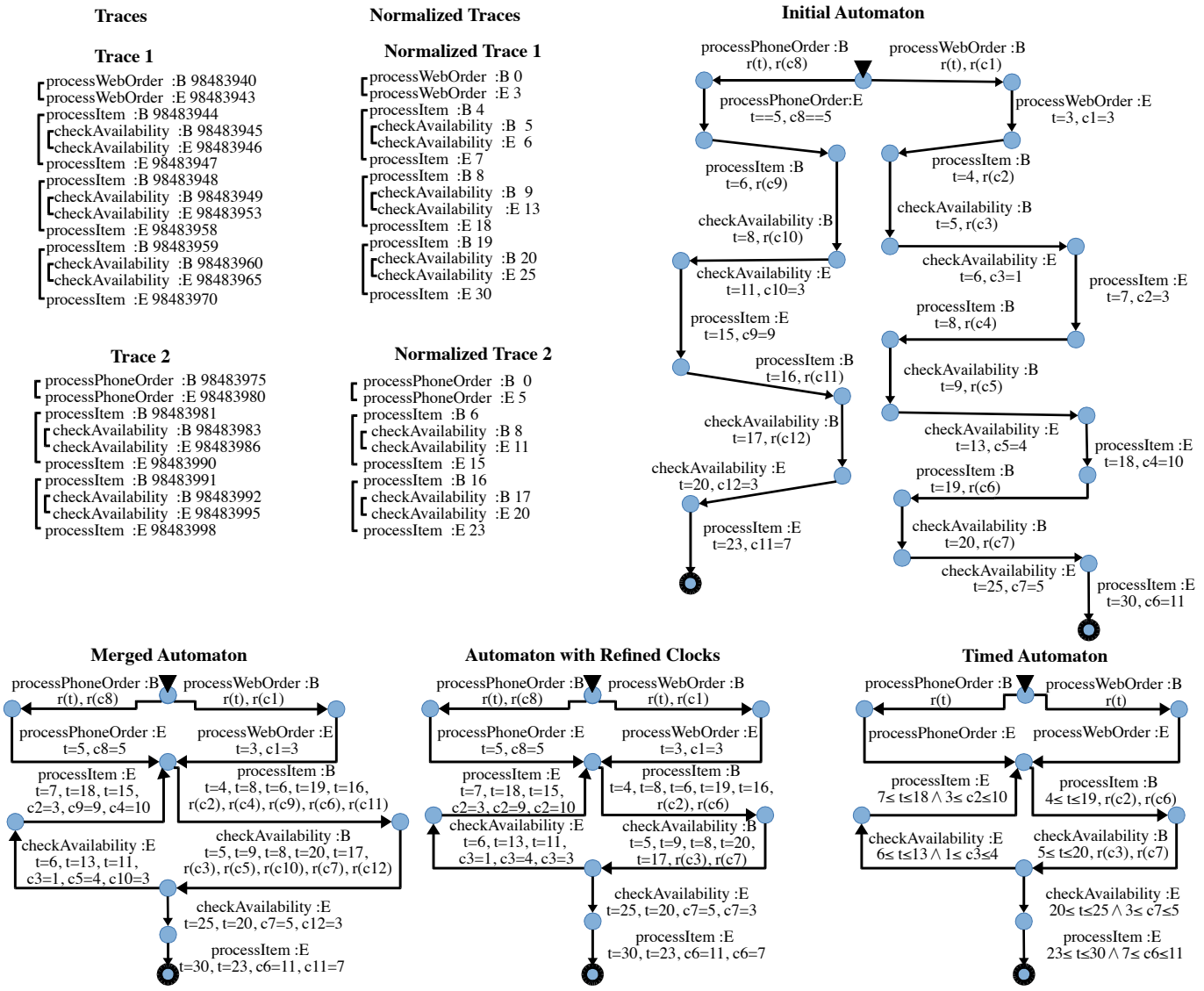


Fig. 1. Inference of a TA with *TkT*: input traces, intermediate results, and output.

TABLE 1
Functions used to generate the *Initial Timed Automaton*.

Operation	Description
$event_i$	Is a tuple $(type_i, op_i, time_i)$ that represents the event with its type $type_i$, operation name op_i and timestamp $time_i$.
$type(event_i)$	Returns the type $type_i$ of the event $event_i$.
$op(event_i)$	Returns the name of the operation op_i performed by the event $event_i$.
$time(event_i)$	Returns the timestamp $time_i$ of the $event_i$.
$pair(event_i)$	Returns the event $event_x$ which has as type the opposite of $event_i$, e.g., if $type(event_i) = B$ then $type(event_x) = E$.
$clock(event_i)$	Returns the clock that counts the duration of the operation in $event_i$. This is well defined only in the generation of the initial automaton where each operation instance is associated with a dedicated clock.

by mapping each trace to an independent branch of the automaton.

Given a set of timed traces T, the *Automaton Initialization* step creates a timed automaton defined as follows (Table 1 includes a summary of the definitions useful to understand the algorithm):

- $S = \{s_0\} \cup \{$a state $s_{i,j}$ for each $event_i$ in $trace_j\}$.
- C=$\{$the absolute clock $t\} \cup \{$a relative clock $c_{i,j}$ for each $event_i$ of type B in any $trace_j\}$. Given an event e of type B, we indicate with $clock(e)$ its relative clock.
- $\Sigma = \bigcup_{trace \in T} \bigcup_{(type,op,time) \in trace} op$
- for each $event_i = (type_i, op_i, time_i)$ occurring in a trace $trace_j \in T$, *TkT* adds a transition tr to TR defined as follows $tr = (s_{i-1,j}, op_i, type_i, g_i, r_i, s_{i,j})$, where
  - $s_{0,j}$ is the initial state s_0 for any value of j,
  - g_i consists of two equalities $t = time_i$, and $clock(event_i) = time_i - time(pair(event_i))$. The

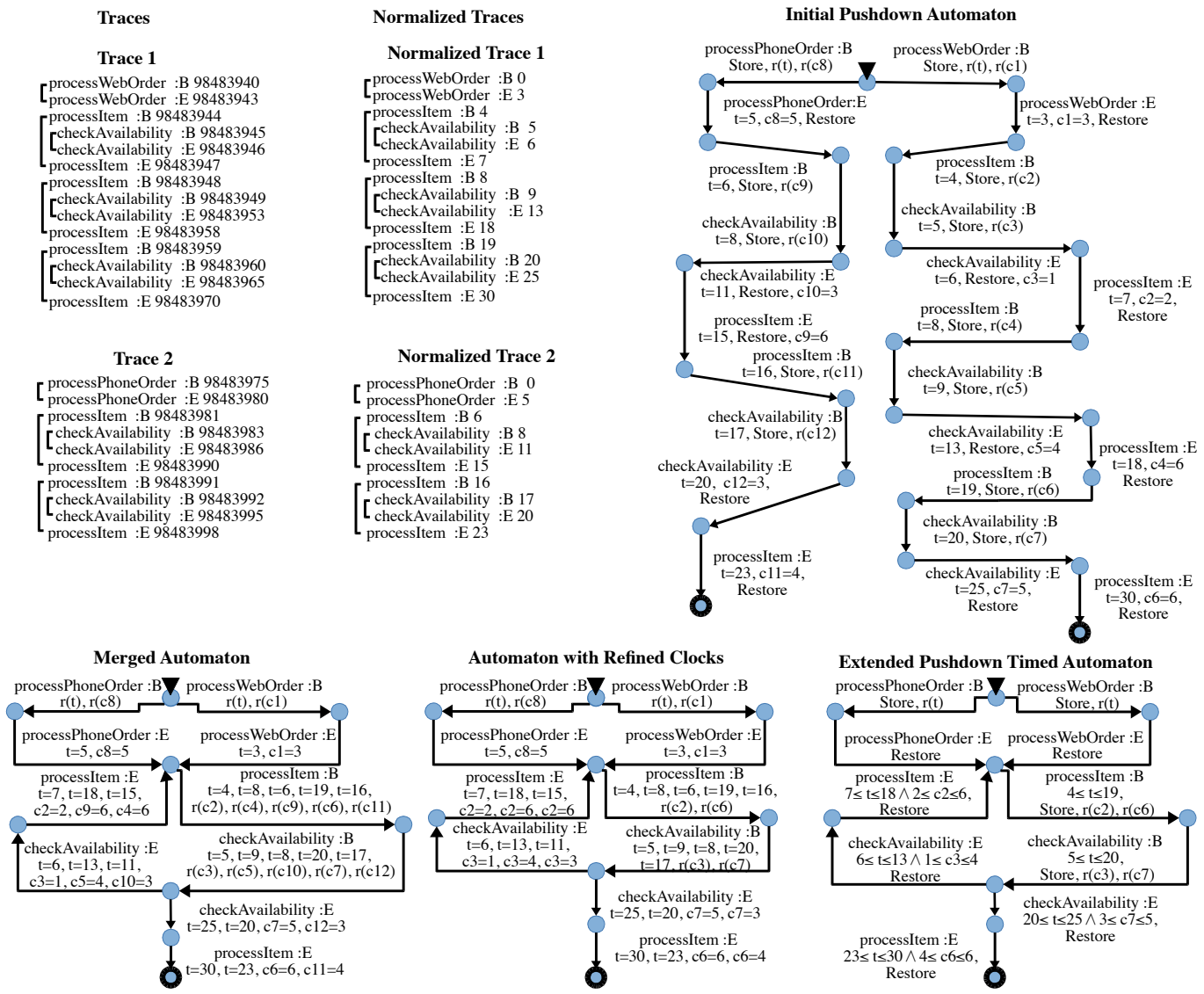


Fig. 2. Inference of a EPTA with *TkT*: input traces, intermediate results, and output.

second equality constrains the value of the clock associated with $event_i$ and is present only if $type_i = E$.

– if $type_i = E$, $r_i = \varnothing$ (clocks are reset only when operations start); otherwise if $type_i = B$, if $i > 1$ then $r_i = \{c_{i,j}\}$, else $r_i = \{t, c_{1,j}\}$ (the clock measuring absolute time is reset on the first transition only).

3.2.2 Initial Pushdown Automaton

TkT can also generate an *Extended Pushdown Timed Automaton* where the clocks used to measure the time spent in an operation (e.g., a method) do not consider the time spent in the execution of the nested operations (e.g., nested method invocations). At the model level, this is obtained by inferring an EPTA where every event of type B produces a transition with a *Store* action, which suspends the existing clocks (with the exception of the global clock, if any), and every event of type E produces a transition with a *Restore* action, which resumes the clocks suspended by the previous Store operation to the values they had when they were suspended. Inferring an EPTA instead of a TA can be a better option when the software has a strongly cyclic behavior. Indeed, the duration of an operation with a loop of nested operations would not be affected by the number of repetitions of the loop. This case is empirically investigated in RQ6 discussed in Section 4.

To build an EPTA that reflects the characteristics described above, *TkT* first builds an *Initial Pushdown Automaton* that accepts all an only the executions stored in the traces and that contains *Store* and *Restore* activities.

In particular, given a set of timed execution traces T, the *Automaton Initialization* step creates an initial pushdown automaton that matches the initial timed automaton described above except for transitions, which present different guards and include *Store/Restore* operations. We describe below how transitions $tr = (s_{i-1,j}, op_i, type_i, g_i, r_i, stackOp_j, s_{i,j})$ are defined in the initial pushdown automaton:

- $s_{0,j}$ represents the initial state s_0 for any value of j,
- g_i consists of two equalities $t = time_i$ and $clock((event_i)) = duration_i$. The value $duration_i$ is defined as the duration of the considered operation (i.e., $time_i - time(pair(event_i))$) minus the duration of the nested operations ($\sum(time_j - time(pair(event_j)))$ for all $event_j$ corresponding to operations completed between $pair(event_i)$ and $event_i$). The second equality constraint is present only when $type_i = E$.
- If $type_i = E$ then $r_i = \emptyset$, because the clocks are only reset when the operations start. If $type_i = B$ and $i > 1$ then $r = \{c_{i,j}\}$, otherwise $r = \{t, c_{1,j}\}$ because the clock measuring absolute time is reset on the first transition only (as done in the *Initial Timed Automaton*).
- If $type_i = E$ then $stackOp_j = Restore$ because the termination of an operation triggers a restore of the clocks.
- If $type_i = B$ then $stackOp_j = Store$ because the beginning of a new operation suspends the existing clocks.

Note that at this point, for both timed and extended pushdown automata, each instance of a (same) operation is associated with a different relative clock. For instance, the duration of the first occurrence of the `processItem` operation in the *Normalized Trace 1* is measured by clock `c2`, while the second occurrence of the same operation in the same normalized trace is measured by clock `c4`, in both automata (see Figures 1 and 2). The automata also define an *absolute clock* `t` that is reset when the first operation is executed and checked at every transition of the model with an equality constraint.

3.3 State Merging

The *State Merging* step iteratively merges all states that accept the same sequences of events until no more states can be merged. This is a standard strategy introduced in the k-Tail algorithm [36], and also exploited by several other algorithms [10], [16], [17], [40], to produce models that generalize a set of observations. *TkT* defines a version of the state merging process that can handle timing information for both timed automata and pushdown automata.

The state merging process starts with the computation of the kFuture of each state, that is, the set of all event sequences of maximum length k that can be accepted by a given state. Since *TkT* takes the type of event (i.e., whether an event indicates either the beginning or the end of an operation) into consideration, the kFuture includes this information.

For example, if k is equal to 2, the kFuture of the initial state of the *Initial Automaton* shown in Figure 1 and the *Initial Pushdown Automaton* shown in Figure 2, is composed of two sequences of length 2. The first sequence consists of the beginning followed by the end of the execution of the `processWebOrder` operation. The second sequence consists of the beginning followed by the end of the execution of the `processPhoneOrder` operation.

When two states have the same kFuture, they are assumed to represent a same state of the software system, and are thus merged into a single state. This process may change the source and the target states of several transitions, and consequently there might be created redundant transitions that can be merged into a single transition. Two transitions are *redundant* if they start from the same state, end into the same state, and have the same event and type. More formally, given two transitions $tr = (s_a, event, type, sop, g, r, s_b)$ and $tr' = (s'_a, event', type', sop', g', r', s'_b)$, they are redundant if $s_a = s'_a$, $s_b = s'_b$, $event = event'$, $type = type'$ and $sop = sop'$. The merge process drops both transitions and adds a single transition annotated with the values from the two dropped transitions. More formally, the transition added after dropping tr and tr' is $tr_{merged} = (s_a, event, type, sop, g \cup g', r \cup r', s_b)$, essentially the resulting transition accumulates the guard conditions and the reset operations present in the merged transitions. We reported the definition for transitions in the pushdown automaton. The definition for the transitions in the timed automaton is the same, ignoring the operation sop.

We call the automaton resulting from this step *Merged Automaton*. Figures 1 and 2 show the *Merged Automaton* obtained from the *Initial Automaton* and the *Initial Pushdown Automaton* by applying the state merging process with $k = 2$.

3.4 Clock Refinement

The *Merged Automaton* may accept more behaviors than the *Initial Automaton* and the *Initial Pushdown Automaton*. However, up to this point, the generalization has only targeted the sequences of events that can be accepted by the automaton, while the timing information still overfits the information in the input traces. For instance, the guard conditions still consist of sets of equality constraints on relative and absolute clocks. It is thus important to generalize the time observations to allow a proper degree of flexibility, so that the model can be used to discover behavioral anomalies on the timing aspect, without being too sensitive to noise.

To estimate the time that might be taken by an operation to complete, it is important to exploit as many observations as possible. Since *TkT* creates a relative clock for each occurrence of each operation in the traces, there is exactly one observation for each relative clock in the model, which is not enough to distill any general information about the timing of the operations. However, the *State Merging* step produces *redundant clocks* that measure the duration of exactly the same behaviors. For instance, clocks $c2$, $c4$, and $c9$ in the *Merged Automaton* in Figures 1 and 2 measure the duration of the `processItem` operation. These individual observations over different relative clocks can be simplified into multiple observations of a same relative clock, which is the starting point for distilling more general information about the duration of the operations. The *Clock Refinement* step performs this simplification over the redundant clocks.

In particular, we define two relative clocks c_a and c_b to be redundant if they are both reset and checked on the same transitions. Note that, by construction, each relative clock is reset on exactly one transition and is checked with a guard condition on exactly one transition. Thus, if $reset(c)$ is the transition where the relative clock c is reset, and $check(c)$ is the transition where the relative clock c is checked with a guard condition, two clocks c_a and c_b are redundant if

$reset(c_a) = reset(c_b)$ and $check(c_a) = check(c_b)$. This strategy to identify and remove redundant clocks can be seen as a special case of the clock reduction strategy based on equality between clocks defined by Daws and Yovine [41].

TkT simplifies redundant clocks by dropping one of the two clocks, namely c_b, and its reset operation, and renaming all the occurrences of the dropped clock in its guard condition with the redundant clock that survives, namely all occurrences of c_b are replaced with c_a. This transformation is performed for all redundant clocks producing an *Automaton with Refined Clocks*, as shown in Figures 1 and 2. For instance, clocks *c2*, *c4*, and *c9* are all renamed as clock *c2*. The refined automata use fewer clocks than the *Initial Automaton* and the *Initial Pushdown Automaton*, with each clock being associated with multiple observations. Note that the *Automaton with Refined Clocks* are intermediate representations where clock constraints are simply accumulated, in contrast with the final TA and EPTA automata where all the constraints on a transition must be satisfied to take a transition.

Clock refinement is not applied to the clock that measures the absolute time because it cannot be redundant with any other clock. Note that the state merging process is already sufficient to accumulate multiple observations on each transition for the clock measuring the absolute time, contrarily to the relative clocks that need this refinement step. In fact, no relative clock can play the role of an absolute clock because each relative clock measures the duration of a specific operation, while the absolute clock measures at what time any operation may end with respect to the beginning of the execution.

3.5 Guards Generation

The *Guards Generation* step iterates on the transitions of the *Automaton with Refined Clocks* and applies a *guard generation policy* to the data available on each transition to produce the *guard* of the transition. The guard generation policy is a function that takes a set of equality constraints on a same clock as input and generates a guard on that same clock as output. When the data associated with a transition refer to multiple clocks, the policy is also applied multiple times, once on each subset of equality constraints on a same clock. For instance, when the set of equality constraints {`t`=30, `t`=23, `c6`=11, `c6`=7} associated with the transition labeled `processItem` in the *Automaton with Refined Clocks* shown in Figure 1 is processed, the guard generation policy is applied twice, once on the data about clock `t` (i.e., {`t`=30, `t`=23}) and once on the data about clock `c6` (i.e., {`c6`=11, `c6`=7}). The generated guards are combined with conjunctions.

We defined two guard generation policies: the *min-max ϵ-policy* and the *γ-confidence policy*. Given a set of observations for a clock `c`, the min-max ϵ-policy generates a guard that constrains the value of `c` to the interval $[(1-\epsilon)min, (1+\epsilon)max]$, where min and max are the minimum and maximum values of `c` in the input values, and ϵ is a value in the range $[0, 1]$. Small values of ϵ bound the interval to the range of values provided as input, while large values of ϵ produces a more flexible interval, up to $[0, 2max]$ in case $\epsilon = 1$. Figures 1 and 2 show, respectively, the *Timed Automaton* and the *Extended Pushdown Timed Automaton* generated with the min-max 0-policy.

The *γ-confidence policy* generates a confidence interval that has a cumulative probability being equal to γ to include the possible duration of an operation based on the collected samples and assuming a normal distribution of the durations [42]. We consider $\gamma = 0.95$ and $\gamma = 0.99$ as possible values of the parameter. If the interval generated by the *γ-confidence policy* does not include all the values observed for a clock (this may sometimes happen due to outliers in the values used to compute the interval), *TkT* extends the interval until including the min/max values of the clock. Although the need to extend intervals rarely happens in practice, it is necessary to formally guarantee that the inferred model accepts every input trace.

If only one value has been observed for a clock, both policies do not generate any guard and the original constraint is dropped.

Note that *TkT* generates timed automata that accept every timed trace used for the inference by construction. In fact both the *Initial Automaton* (with the corresponding TA) and the *Initial Pushdown Automaton* (with the corresponding EPTA) accept exactly the timed traces provided as input. The state-merging process may only increase the combination of events that might be accepted by the model and cannot drop any behavior. The *Clock Refinement* step simplifies the model without altering the set of accepted behaviors. Finally, the *Guards Generation* step can only increase the range of acceptable timing for the events in the traces.

3.6 Complexity Analysis

TkT extends k-Tail with the *Clock Refinement* and the *Guards Generation* steps. The other steps of the algorithm are the same as the steps in k-Tail, with the only addition of operations to annotate transitions with resets and checks on the clocks. In the worst case, the number of added resets and checks is of the order of n, where n is the cumulative length of all traces.

The number of clocks introduced by *TkT* in the initial automaton is $\lceil \frac{n}{2} \rceil + 1$, that is, one clock for each pair of events plus the global clock. This is also the maximum number of clocks that can be refined in the *Clock Refinement* step. *Clock Refinement* is performed by comparing the sets of clocks being reset or checked in every pair of transitions of the TA, which leads to $\mathcal{O}(n^2)$. Finally, *Guards Generation* iterates over all the transitions to transform the samples into constraints. Since the number of transitions cannot be more than n, the cost of this step is in the order of n.

Since n^2 is negligible with respect to the complexity of k-Tail, which is $\mathcal{O}(n^2 \times |A|^k)$ [16], [43] , where $|A|$ is the size of the alphabet of the automaton, and k is the parameter of the algorithm, the complexity of *TkT* remains the same of k-Tail. Indeed, the dominating cost factor is still the state merging process, which is the same in the two algorithms.

3.7 Usage Scenarios

TkT is a specification mining technique that can be used to mine models when they are not available. Similarly to other specification mining techniques [9], [16], [19], [20], [21], [40], [44], [45], *TkT* only requires the availability of positive samples, that is, samples of behaviors that must

be accepted by the inferred model. In this case, it is the responsibility of the developer to run the software under analysis and make sure that the resulting executions do not include failures. These executions can be obtained in many ways, for instance by running a set of (passing) test cases or by running the system against an oracle. Note that in the latter case the oracle may refer to the end-to-end behavior of the system, while the models can be inferred from the individual components of the system. For example, the oracle may check the correctness of the output produced by a purchase procedure, while timed automa that represent the behavior of the individual components of the system when the purchase procedure is executed can be inferred. Alternatively, the oracle may also directly discriminate the correctness of the traces that can be then used to infer a model [46].

Inferred models, including timed automata inferred by *TkT*, can support a variety of tasks. For instance, the tester can use the inferred models to derive new test cases that are different from the ones used in the inference process [7], [8], [47]. Alternatively, when a failure is detected, using either an automatic or manual oracle, models representing the behavior of the individual components can be used to support debugging [9], [10], [11], [12], [13], that is, a failure trace can be compared against the inferred models to identity the anomalous events likely responsible for the failure. Investigating these possible applications when the timing and not only the functional dimension is relevant is part of our future work.

4 Empirical Evaluation

To empirically evaluate *TkT*, we investigated the following seven research questions.

RQ1: Is *TkT* able to infer generic models that *comprehensively* capture the behavior of the software? RQ1 investigates the sensitivity of *TkT* with TAs and with EPTAs, that is, its capability to derive models that accept legal traces, including the ones that have not been used for the inference, compared to the Perfume state-of-the-art technique.

RQ2: Is *TkT* able to infer precise models that *reject* anomalous software behaviors? RQ2 investigates the specificity of *TkT* with TAs and with EPTAs, that is, its capability to derive models that reject anomalous traces, compared to the Perfume state-of-the-art technique.

RQ3: How does *TkT balance* the ability to deal with legal and illegal behaviors? RQ3 investigates how *TkT* with TAs and with EPTAs balances sensitivity and specificity, compared to Perfume.

RQ4: How does *absolute time* affect the effectiveness of *TkT*? RQ4 investigates how the generation of a clock that measures the absolute time affects sensitivity and specificity.

RQ5: How does *TkT scale* with the size of the traces? RQ5 investigates how the performance of *TkT* with TAs and with EPTAs scales with the number of events that are processed, compared to Perfume.

RQ6: How do EPTAs affect the sensitivity of *TkT*? RQ6 investigates if EPTAs, which rely on relative clocks that ignore the time spent in nested calls, may increase the sensitivity of *TkT*, especially in scenarios where the traces used for the validation include many loop iterations.

RQ7: How does *the number of traces* affect the size of the models inferred by *TkT*? RQ7 investigates to what extent the number of traces affects the size of the models inferred by *TkT*.

4.1 Prototype and Experiment Setup

The *TkT* prototype that we used for the experiments is implemented in Java and is available at https://github.com/lta-disco-unimib-it/tkt. Our implementation supports all the configurations and parameters described in this paper.

There are two main parameters that may influence how the timing behavior is encoded in the inferred models. The first parameter controls if guards on the absolute clock are generated. The second parameter controls the specific guard generation policy that *TkT* must use to generate guards, among the *min-max ϵ-policy* and the *γ-confidence policy*. Both policies can be customized according to a parameter that can be assigned with real values in the range $[0..1]$.

In our empirical evaluation, we study the impact of absolute clocks and both guard generation policies by covering a range of values for their parameters. In the min-max ϵ-policy, we cover the whole domain, sampling more densely the domain for small values of ϵ, where we expect the technique to be more sensitive to changes. In the γ-confidence policy, we consider γ equals to 0.95 and 0.99, to focus on guards that may cover the possible durations with high confidence. These configurations are investigated both with and without the generation of the guards on the absolute clock. Table 2 summarizes the set of configurations considered in our evaluation. Each configuration has an identifier.

The behavior of *TkT* is also influenced by the choice of the parameter k, which does not play a role in guards generation but affects the identification of the states to be merged. In our empirical evaluation, we focus on the parameters that may influence the timing behavior, which is the novel aspect introduced in *TkT* compared to k-Tail, and we do not study the impact of k in the state merging process which has been already studied in many other papers [4], [11], [16], [17], [40], [48]. We rather take advantage of these results to fix the value of k to 2, which has demonstrated to be a good choice when analyzing traces collected from software systems.

4.2 Subjects of the Study

The study has been conducted on two sets of Java programs that enabled us to perform a complementary study on the type of anomalies to be detected by *TkT*, namely, slowdown caused by the execution environment and performance faults.

The first set of programs consists of the implementation of four well-known *algorithms*: the *merge sort* sorting algorithm [49], the *Rabin Karp* pattern matching algorithm [50], the *LZW* compression algorithm [51], and the *LZWdecom* decompression algorithm [51]. We used these algorithms to evaluate the capability of *TkT* and Perfume to discriminate the valid and the invalid executions caused by *anomalies in the execution environment* (e.g., due to overloaded resources). To study this capability, we logged the execution of all the methods in the classes that implement the algorithms both

TABLE 2
TkT configurations.

ID		Type of	Policy	ϵ/γ
abs clock	no abs clock	Automata		
TM1	TM9	TA	min-max	0.05
TM2	TM10	TA	min-max	0.10
TM3	TM11	TA	min-max	0.15
TM4	TM12	TA	min-max	0.20
TM5	TM13	TA	min-max	0.25
TM6	TM14	TA	min-max	0.50
TM7	TM15	TA	min-max	0.75
TM8	TM16	TA	min-max	1.00
EM1	EM9	EPTA	min-max	0.05
EM2	EM10	EPTA	min-max	0.10
EM3	EM11	EPTA	min-max	0.15
EM4	EM12	EPTA	min-max	0.20
EM5	EM13	EPTA	min-max	0.25
EM6	EM14	EPTA	min-max	0.50
EM7	EM15	EPTA	min-max	0.75
EM8	EM16	EPTA	min-max	1.00
TG1	TG3	TA	γ-confidence	0.95
TG2	TG4	TA	γ-confidence	0.99
EG1	EG3	EPTA	γ-confidence	0.95
EG2	EG4	EPTA	γ-confidence	0.99

when the system is not overloaded and when it is overloaded by four other processes intensively executing I/O operations. For each algorithm we produced 100 valid and 100 invalid traces. Our test cases cover more than 90% of the instructions implemented by the algorithms (we ignored inputs triggering exceptional cases). We used EclEmma [52] to measure code coverage.

The second set consists of 12 versions of three well known opensource libraries, the *Google Guava library* [53] (five versions), the *Apache Commons Collections library* [34] (four versions), and the *Apache Commons Math Libraries* [35] (three versions), all affected by different performance problems. For our empirical study we selected the most recent and replicable performance faults reported in the issue trackers of the three libraries at the time of the study, for a total of 12 faults considered in our experiment. Table 3 shows the details of the subjects of our study, column *Library* reports the library name, column *Version* reports the library version affected by the fault, column *Bug Id* reports the bug identifier, column *URL* reports the Web URL from which the bug report can be inspected online. Column *Code Coverage* reports the percentage of instructions of the faulty class (i.e., the class for which we generate traces) that are covered by the test cases considered in our experiments.

We used these library versions to evaluate the capability of *TkT* and Perfume to discriminate the valid executions from the invalid executions caused by *performance faults*. To study this capability, for each version of the library we generated the traces by running the original test suite distributed with the libraries and a non-regression test suite that we implemented to extensively sample the fixed functionality with various inputs. We obtained the valid traces by running the test cases on the fixed version of the program (we extracted the fixes from the version history of the software). We obtained the invalid traces by executing the same test cases on the faulty version of the software.

Since not every test execution of the faulty program necessarily produces a performance failure, we used the overhead observed when running the test suite of the program to objectively discriminate between valid and invalid traces (i.e., failed executions). In particular, the traces recorded during executions showing an overhead smaller than the one observed while running the test suite of the program have been classified as valid, consistently with the interpretation of the developers who have not recognized the performance problems while running the test suites of the subject programs. Traces recorded in executions showing a higher overhead have been classified as invalid. Depending on the library version, this strategy enabled us to generate between 340 and $1,000$ valid traces and between 51 and 501 invalid traces caused by actual performance faults.

In the evaluation, we recorded the traces using an AspectJ advice [54] that intercepts both method entry and exit events.

4.3 RQ1: Is *TkT* able to infer generic models that *comprehensively* capture the behavior of the software?

Procedure: Research question RQ1 investigates the *sensitivity* of *TkT*, that is, its capability to infer models that can accept the legal traces not occurring in the set of traces used for the inference. We study the sensitivity of *TkT* when generating both TAs and EPTAs. Hereafter, we use the acronyms TkT_{TA} and TkT_{EPTA} to refer to the configurations of *TkT* used to infer TAs and EPTA, respectively. Since the quality of the inferred models depends not only on the algorithm, but also on its configuration and the completeness of the traces used for the inference, we studied the effectiveness of *TkT* for all the 40 configurations listed in Table 2 and for sets of traces of various sizes. We performed the experiment for the four algorithms and the 12 versions of the three libraries (Google Guava, Apache Commons Collections, and Apache Commons Math).

In order to compare *TkT* to Perfume, we also executed Perfume against the traces collected from the four algorithms. We do not have results about Perfume applied to the traces collected from the Guava library because Perfume has not been able to infer a model after 10 hours of computation, which was our time limit for each experiment, in contrast to *TkT* that completed every inference task with less than 5 minutes (see RQ5 for a discussion about scalability).

To measure the impact of the number of traces on the sensitivity of the algorithms, we executed TkT_{TA}, TkT_{EPTA}, and Perfume on random subsets of traces, considering from 10% to 100% of the set of valid traces. To mitigate the effect of randomness, we repeated the extraction process 10 times and reported average values. Note that these subsets of traces may miss many valid application behaviors facilitating the generation of incomplete models. More precisely, since different test cases exercise different usage scenarios of the systems under test, the selection of a subset of traces may enable us to study the effects of a test suite with limited effectiveness (e.g., limited code coverage).

For each set of traces, we used the 10-fold cross validation method to measure the sensitivity of TkT_{TA}, TkT_{EPTA}, and Perfume. The 10-fold cross validation method works by dividing an initial set of traces in 10 sets of the same size (folds), using 9 sets for the inference of the model (training

TABLE 3
Libraries case studies: performance faults considered in our analysis.

Library	Version	Bug Id	URL	Code Coverage
Guava	1.0	371	https://github.com/google/guava/issues/371	85.6%
Guava	12.0	1013	https://github.com/google/guava/issues/1013	100.0%
Guava	13.0	1155	https://github.com/google/guava/issues/1155	96.0%
Guava	13.0	1196	https://github.com/google/guava/issues/1196	96.7%
Guava	13.0	1197	https://github.com/google/guava/issues/1197	94.3%
Commons Collections	3.2.1	407	https://issues.apache.org/jira/browse/COLLECTIONS-407	85.3%
Commons Collections	3.2.1	413	https://issues.apache.org/jira/browse/COLLECTIONS-413	100.0%
Commons Collections	3.2.1	425	https://issues.apache.org/jira/browse/COLLECTIONS-425	98.5%
Commons Collections	4.0	534	https://issues.apache.org/jira/browse/COLLECTIONS-534	100.0%
Commons Math	1.0	248	https://issues.apache.org/jira/browse/MATH-248	82.7%
Commons Math	3.0	1153	https://issues.apache.org/jira/browse/MATH-1153	92.1%
Commons Math	3.0	1220	https://issues.apache.org/jira/browse/MATH-1220	97.7%

set) and 1 set for the validation of the model (validation set). The sensitivity of the algorithm is computed as the fraction of the traces in the validation set accepted by the model inferred from the training set. The process is repeated 10 times, every time using a different fold as validation set. The final score is obtained as the average value of the sensitivity for the 10 folds. This entire process is repeated 5 times to mitigate the effect that randomly partitioning a set of traces into 10 folds may have. Overall, the study on the sensitivity required the generation of 41 models (40 for *TkT* and 1 for Perfume) from each training set, for each subset of traces, for each subject application (except for library cases, where Perfume did not complete), and for 5 repetitions, for a total of more than $180,000$ models inferred.

We believe that the number of traces considered for model inference can capture the effect of the quality of test suites on the inferred models.

Results: The boxplots in Figure 3[1] show the sensitivity of the models inferred using TkT_{TA}, TkT_{EPTA}, and Perfume when an increasing number of traces is available for the inference. In particular, each set of consecutives 10-boxes represents a different configuration and each vertical box shows the average fraction of accepted traces among the subject applications.

For both the min-max ϵ-policy and γ-confidence policy we report the configurations with the lowest (*TM1* and *TG1* for TkT_{TA}, *EM1* and *EG1* for TkT_{EPTA}) and the highest sensitivity values (*TM16* and *TG4* for TkT_{TA}, *EM16* and *EG4* for TkT_{EPTA}). We also report the sensitivity of Perfume.

The results show that the sensitivity of both TkT_{TA} and TkT_{EPTA} is high, with values quickly growing beyond 90%. Both policies perform well, with the min-max ϵ-policy performing slightly better than the γ-confidence policy in the case of the algorithms. This is probably due to the availability of a good number of traces that broadly covers the valid behaviors, which can be better exploited by the min-max ϵ-policy.

Both TkT_{TA} and TkT_{EPTA} outperform Perfume in terms of sensitivity. Perfume rejects many traces due to the presence of constraints that overfit the input samples used for the inference. For instance, Perfume never reaches a score of 0.8 for the sensitivity even when using 90% of the available traces for the inference, while the min-max ϵ-policy already achieves higher values of sensitivity when 18% of the available traces is used for the inference. Based on a non-parametric Mann Whitney test, the difference in accuracy between each *TkT* configuration and the corresponding Perfume configuration (e.g., TM1 with 9% of traces in the training set compared to Perfume with 9% of traces in the training set) is always significant with a p-value < 0.05.

Finally, we note that each group of boxplots for *TkT* in Figure 3[1] follows a similar pattern, with increasing median and decreasing inter-quartile range, for a higher portion of traces used for the inference. This result indicates that *TkT* converges towards higher quality models (i.e., better sensitivity) when increasing the number of traces. Models without global clocks (i.e., *TM* and *EM*) achieve better results. The same pattern cannot be seen in the case of Perfume. Indeed, the interquartile range remains almost the same for different portions of traces used for the inference (i.e., all the experiments show the same degree of variability). Also, the median is not above the value 0.6 even when more than 50% of the available traces is used for inference. We can conclude that Perfume led to models that tend to overfit the training set and are unlikely to accept new, unseen traces.

4.4 RQ2: Is *TkT* able to infer precise models that *reject* anomalous software behaviors?

Procedure: Research question RQ2 investigates the *specificity* of *TkT*, that is, its capability to reject invalid traces. In our experiments invalid traces are characterized by the presence of anomalous timing due to either system overloading or performance faults.

We measure the specificity of a model as the fraction of invalid traces rejected by the model. We studied the specificity for the same range of configurations, size of the training sets, and subject applications used for RQ1.

Results: The boxplots in Figure 4[1] show the specificity of the models inferred with TkT_{TA}, TkT_{EPTA}, and Perfume when an increasing number of traces is available for the inference. In particular, each set of 10-boxes represents a different configuration and each vertical box shows the average fraction of rejected traces among the subject applications.

For both the min-max ϵ-policy and γ-confidence policy we report the configurations with the highest (i.e., *TM1* and

1. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range. Outliers are not printed. Red lines show average.

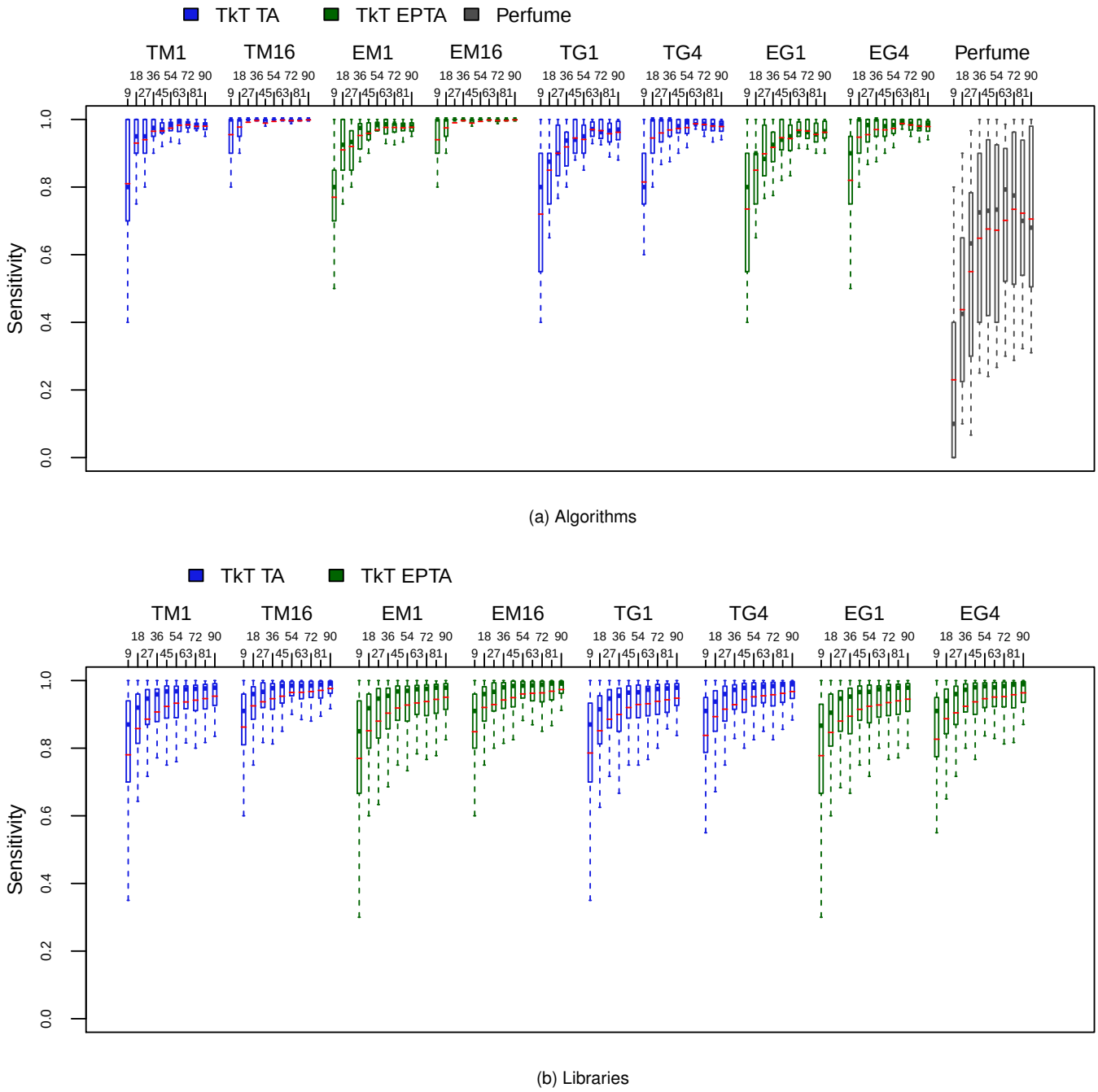


Fig. 3. RQ1: Sensitivity wrt percentage of available traces included in the training set (i.e., 9%, 18%, 27%, 36%, 45%, 54%, 63%, 72%, 81%, 90%).

TG1 for TkT_{TA}, *EM1* and *EG1* for TkT_{EPTA}) and the lowest (i.e., *TM16* and *TG4* for TkT_{TA}, *EM16* and *EG4* for TkT_{EPTA}) specificity values. We also report the specificity of Perfume limitedly to the case of the overloaded environment because it has not been able to infer the models for the libraries.

Perfume achieves a perfect specificity for any number of input traces. This is due to the overfitting of the timing constraints generated by Perfume, which can easily reject any trace that includes a small variation of the timing behavior observed on the traces used for the inference.

TkT, in both cases (i.e., with TAs and with EPTAs), achieves almost the same result than Perfume when applied to the invalid traces obtained from an overloaded environment. Indeed, the best configurations (i.e., *TM1*, and *EM1*) achieve 100% specificity and the worst configurations (i.e., *TM16*, *EM16*, *TG4*, *EG4*) achieve more than 90% specificity. TAs and EPTAs behave differently, with TAs having a slightly better capability to identify invalid traces. Based on a non-parametric Mann Whitney test, the difference in accuracy between each *TkT* configuration and Perfume is significant with a p-value < 0.05 for six configurations (i.e., *TG1*, *EG1*, *TM16*, *EM16*, *TG4*, *EG4*). In two cases (i.e., *TM1*, and *EM1*) there is no difference between *TkT* and Perfume since they both reject all the invalid traces.

The boxplots in Figure 4[1] show that the configurations leading to TAs with the highest sensitivity have the lowest

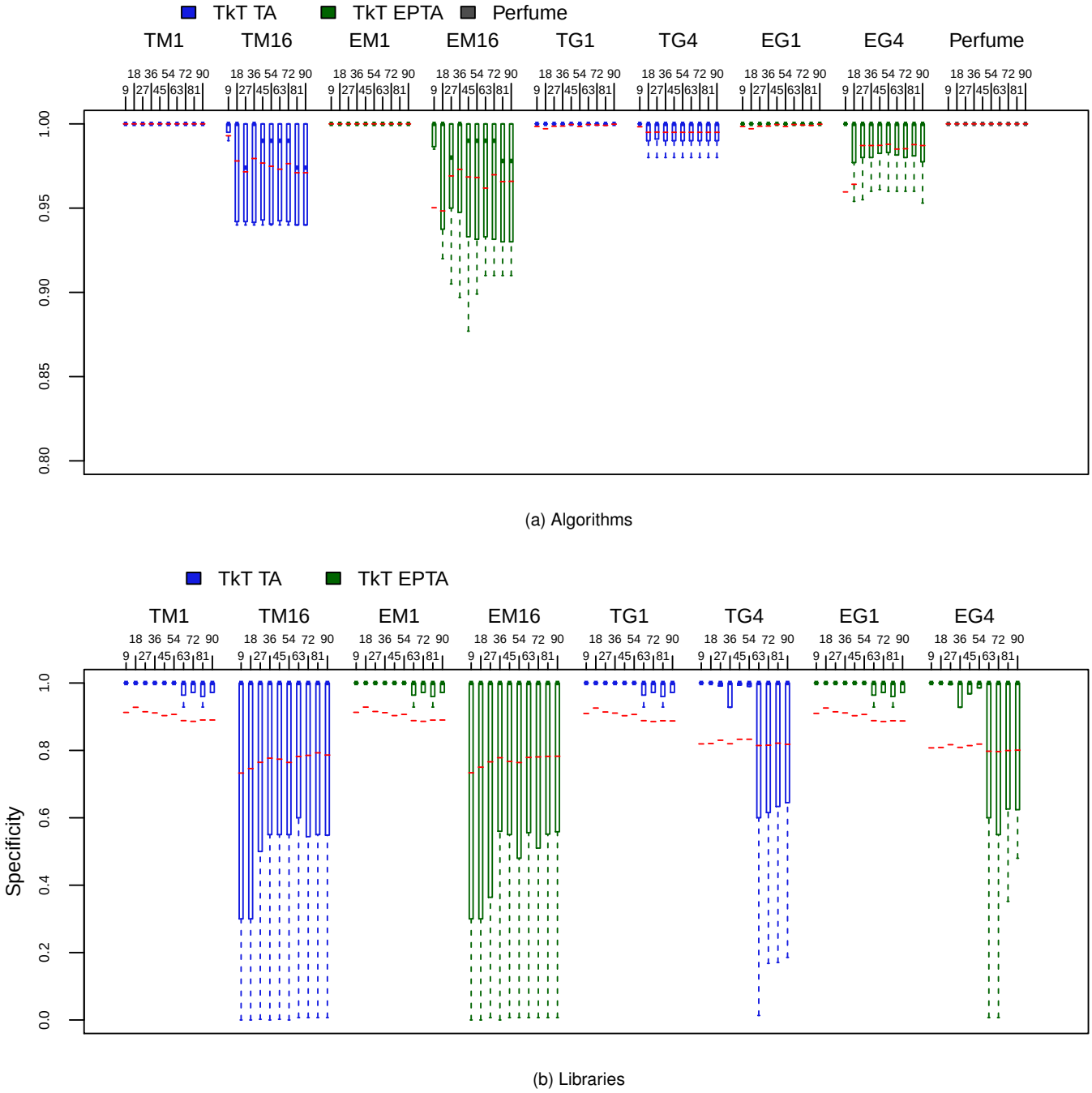


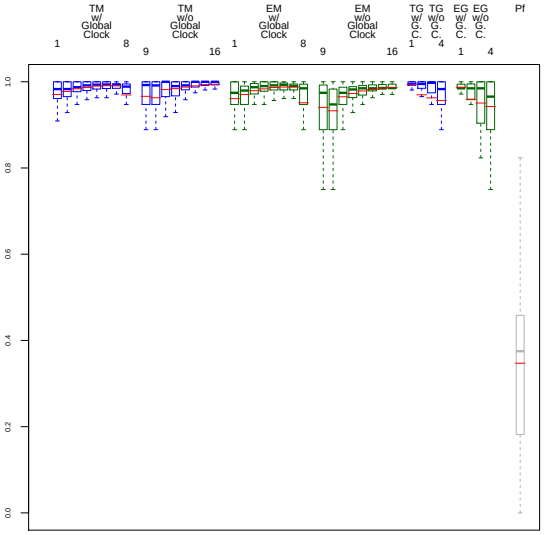
Fig. 4. RQ2: Specificity wrt available traces included in the training set (i.e., 9%, 18%, 27%, 36%, 45%, 54%, 63%, 72%, 81%, 90%).

specificity, independently of the number of traces used for the inference process (the interquartile range is large for all the cases). In terms of guard generation policies, the γ-confidence policy is able to generate invariants better fitting the observations with a higher capability to reject invalid traces. In fact, even the worst configuration (i.e., EG4) achieves nearly perfect results.

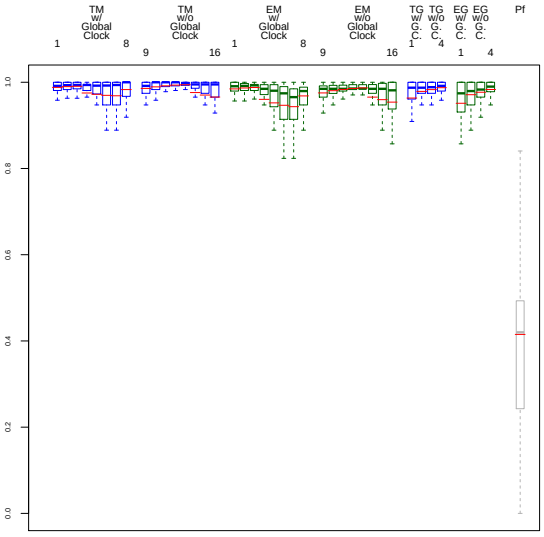
The specificity is lower when applied to performance faults. In particular, the worst configurations (i.e., *TM16*, *EM16*, *TG4*, *EG4*) are much less performing compared to the case of an overloaded environment. However, the top configurations (i.e., *TM1*, *EM1*, *TG1*, *EG1*) still behave extremely well.

We inspected the traces and faults to understand why some configurations do not perform well. This is almost due to a single performance fault (fault number 1196) that produces traces with an extremely small overhead, which can be hardly recognized by TkT_{TA} and TkT_{EPTA}, and could also be hardly recognized by a developer.

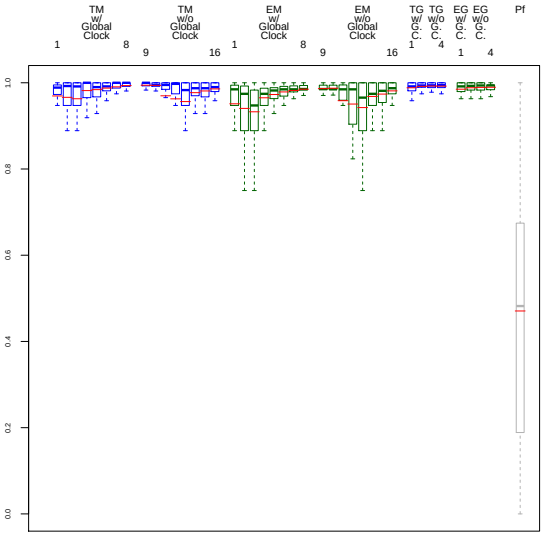
Overall, these results indicate that the models inferred by *TkT* can be very effective in discriminating both classes of invalid executions (overloaded environment and performance faults).



(a) Algorithms with $\%\,Traces \leq 30$

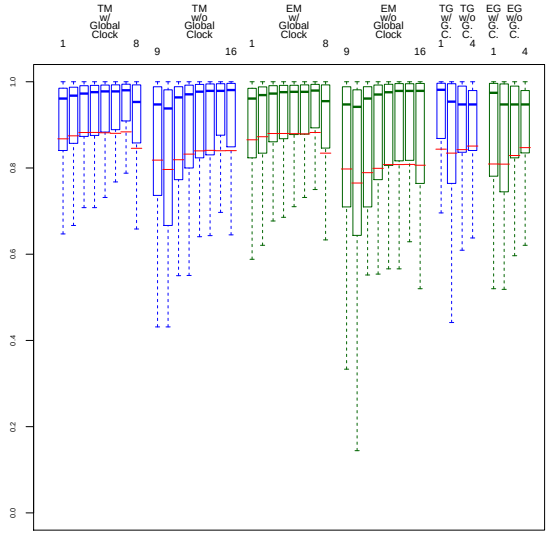


(b) Algorithms with $30 < \%\,Traces < 70$

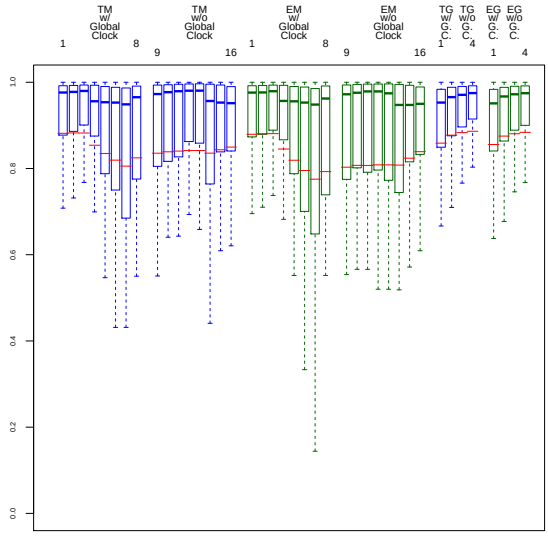


(c) Algorithms with $\%\,Traces \geq 70$

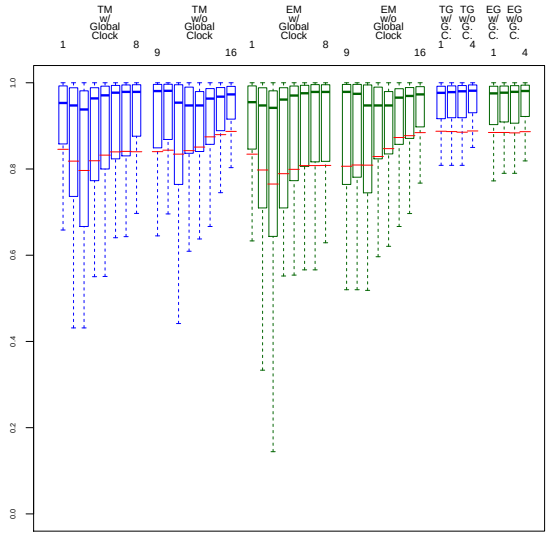
Fig. 5. RQ3: Harmonic mean of specificity and sensitivity wrt available traces included in the training set. Algorithmic cases.



(a) Libraries with $\%\,Traces \leq 30$



(b) Libraries with $30 < \%\,Traces < 70$



(c) Libraries with $\%\,Traces \geq 70$

Fig. 6. RQ3: Harmonic mean of specificity and sensitivity wrt available traces included in the training set. Library cases.

4.5 RQ3: How does *TkT balance* the ability to deal with legal and illegal behaviors?

Procedure: RQ3 investigates how TkT_{TA} and TkT_{EPTA} balance sensitivity and specificity, also in comparison to Perfume. Finding a balance between sensitivity and specificity is important to prevent the generation of models that may trivially reject or accept all the traces. In particular, a good model should tolerate small changes on the timing behavior, promptly reporting any significant deviation.

To study this balance, we computed the harmonic mean of sensitivity and specificity for the configurations that we studied in RQ1 and RQ2. We aggregated data according to the percentage of available traces used for the inference. We distinguished between a small portion of traces (at most 30% of the available traces used for the inference), an intermediate portion of traces (the number of traces used for inference comprised between 30% and 70%), and a high portion (more than 70%) of the available traces.

Results: The boxplots in Figures 5[1] and 6[1] show the results for all the configurations when a small, intermediate, and high number of traces are available.

In both cases (i.e., with TA and with EPTA) *TkT* performed better than Perfume, with larger differences when fewer traces are available. Based on a non-parametric Mann Whitney test, for each setup (i.e., small, intermediate, and high portion of available traces), the difference in accuracy between each *TkT* configuration and Perfume is always significant with a p-value < 0.05.

TkT demonstrated to be relatively sensitive to the choice of the parameters in the investigated configurations. Indeed, values from the middle to the top of the range for the min-max ϵ-policy (i.e., TM4-8, TM13-16) all performed similarly. The choice is relatively difficult also for the γ-confidence policy, which again performed similarly to the min-max ϵ-policy. Small values of γ may work better when few traces are available, while higher value of γ may work better when several traces are available, but again the choice of the parameterization is not critical.

We can also notice that using absolute clocks systematically achieves better results than not using them for all the investigated configurations. This aspect is further discussed with RQ4.

Overall, the results show that *TkT* is not highly sensitive to the choice of the parameterization, with only small values of ϵ for the min-max ϵ-policy that performed slightly worse than the other configurations, and thus should be avoided. Moreover, *TkT* obtained an almost optimal balance between specificity and sensitivity, clearly outperforming Perfume.

4.6 RQ4: How does *absolute time* affect the effectiveness of *TkT*?

Procedure: RQ4 investigates how the use of the absolute clocks affects sensitivity and specificity. The adoption of absolute time may allow to detect operations that begin or terminate too late with respect to the beginning of the trace, but may also introduce subtle dependencies among the operations. To investigate the effect of adopting absolute time, we compare the sensitivity and the specificity of the configurations that use absolute clocks to the configurations that do not use them, for all the available cases.

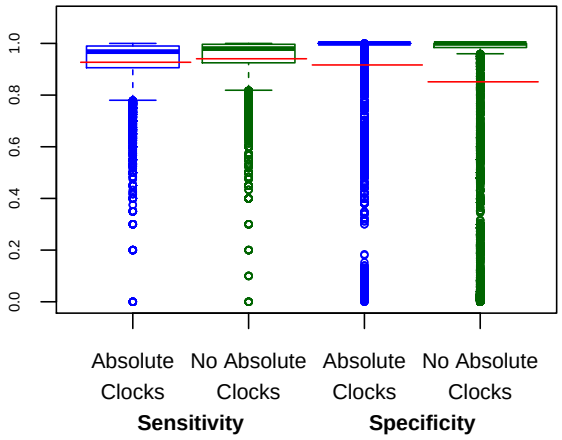


Fig. 7. RQ4: Distribution of sensitivity/specificity with and without absolute clocks. Red lines show the average across all inferred models.

Results: The box plot in Figure 7[2] shows the results for sensitivity and specificity with and without absolute clocks. According to our set of experiments, the adoption of absolute clocks leads to a small decrease of sensitivity while they contribute to a slightly higher specificity.

Results obtained with RQ3 shows that, overall, using absolute clocks might be beneficial for the model. However, if there is a reason to prioritize specificity to sensitivity, or vice versa, the absolute clock might be used or ignored, respectively.

4.7 RQ5: How does *TkT scale* with the size of the traces?

Procedure: To investigate RQ5, we compare the inference time of *TkT* (when it generates TA and EPTA) to the inference time of Perfume. In particular, we investigate how the cost of the inference increases when an increasing number of traces are processed by the algorithms. We measured the time necessary to infer a model for all the executions performed to answer RQ1. The experiments have been run on a Lenovo System X 3500 M5 Server with 32 cores (3.20GHz). Since Perfume did not complete on the library cases, we only considered the four algorithms for this study.

Results: Figure 8[2] shows a boxplot with the inference time of *TkT* compared to Perfume. Each box represents multiple executions all performed with samples of the same size.

Data show that Perfume is significantly slower than *TkT*. For example when the number of traces to be processed is close to 100, *TkT* is almost three orders of magnitude faster than Perfume (Perfume takes on average 34.7 minutes, while *TkT* takes 0.5 seconds). More specifically, the cost of running Perfume increases quickly with respect to the number of processed events, while the cost of running *TkT* growths by a negligible fraction.

The observed differences in performance can find a justification also looking at the complexity of the algorithms.

2. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range. Red lines show average.

Although a formal complexity analysis of Perfume is not available, there are a few steps in Perfume that can make the algorithm slower than *TkT*. Perfume is an extension of *Synoptic* [14], which starts its inference process by mining properties from execution traces. Perfume extends the set of mined properties used by Synoptic employing a total of seven properties. Moreover, the inference process uses model checking to implement a counterexample-guided abstraction refinement process that iteratively manipulates the inferred automaton until reaching a refined model. Finally, Perfume runs k-Tail with k=1 to reduce the inferred model. The total cost of running property mining, model checking, and k-Tail with k=1 is plausibly more expensive than running *TkT* with k=2, whose complexity is of the same order of k-Tail (see Section 3.6).

These results suggest that Perfume might hardly scale to large systems producing long traces, as we experienced with the libraries, contrarily to *TkT* that scaled gracefully with the number of events.

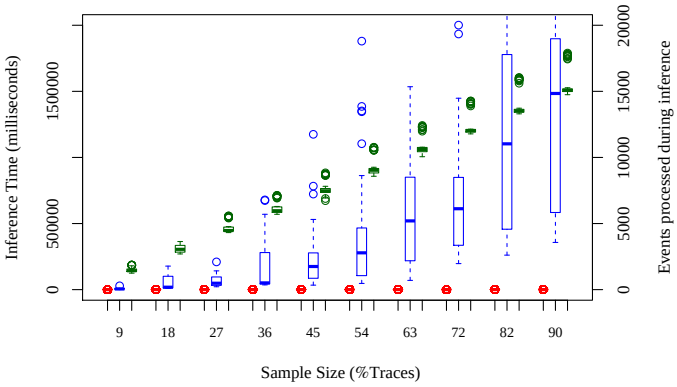


Fig. 8. RQ5: Inference time of *TkT* (red) and Perfume (blue), plus number of events processed during inference (green). To save space, we plot the chart area below 2,000,000 (y axis). The upper whiskers not shown in the figure are the ones for the boxplots generated with 82 and 90 traces and their values are equal to 35,18,000 and 38,06,000, respectively.

4.8 RQ6: How do EPTAs affect the sensitivity of *TkT*?

Procedure: To respond to RQ6, we performed an experiment to compare the sensitivity obtained with TA to the sensitivity obtained with EPTA when the traces used for validation include several repetitions of iterative behaviors not observed in the traces used to infer the models.

To perform this experiment, we considered traces obtained while executing loop iterations many times. More precisely, we considered as subjects of our study components belonging to the case studies in Table 3 whose iterative behavior depends on the size of the input (typically a Java collection class). We selected the classes affected by the Guava bug IDs 371, 1013, 1055 and by the Commons Collections bug IDs 407, 413, 425, and 534. For each of these case studies, we derived 1,000 test cases, each one exercising the software with inputs (typically a collection class) that differ for their size (i.e., the number of elements in the collection).

To perform the experiment we inferred models by selecting traces generated with an input of size up to k, with the value of k ranging from 100 to 1,000 with a step equals to 100. To evaluate the sensitivity of the model, we used the remaining traces. We performed the experiment for all the configurations in Table 2 except the ones including absolute clocks since absolute clocks alone may negatively affect sensitivity.

Results: The boxplots in Figure 9[1] show the distribution of the percentage of valid traces accepted by the models inferred with *TkT* when traces with an increasing length are available for inference. In particular, we report eight groups of boxplots that capture the results achieved with eight different configurations of *TkT*, four related to TA, and four related to EPTA. To ease visual comparison of the results, we display pairs of configurations from left to right, such that each pair differs only for the kind of model derived.

For both the min-max ϵ-policy and the γ-confidence policy we report the configurations with the lowest (i.e., *TM9*, *EM9* , *TM16*, *EM16*), and the highest (i.e., *TG3*, *EG3*, *TG4*, and *EG4*) acceptance rate.

Results show that, expectedly, the use of long traces during inference lead to a greater portion of traces being accepted. Indeed, in these cases, the difference in length between the longest trace used for validation and the longest traces used for inference is minimal. More in general, we notice that EPTAs better tolerate iterations compared to TAs since they produce models with higher sensitivity. This is likely the effect of the main characteristic of the EPTAs models derived by *TkT*, that is, ignoring the time spent in nested calls, which might be large in long executions. For example, configuration *EM16* achieves an average sensitivity of 0.8 even when the length of the traces used for inference is one-tenth of the maximal length of the traces used for validating the model. This may suggest the adoption of EPTAs models to validate long execution traces collected in the field.

4.9 RQ7: How does *the number of traces* affect the size of the inferred models?

Procedure: To address RQ7, we study the size of the models inferred by *TkT* when an increasing number of traces are processed. We consider all the executions performed to answer RQ1. We do not distinguish between TkT_{TA} and TkT_{EPTA} because, by construction, for a same set of traces, they lead to automata with the same number of states and transitions. Indeed, the automata generated by the two algorithms differ only for their guard conditions.

In addition, we measure how the number of traces affects the degree of reduction in the number of clocks included in the model, compared to the number of clocks in the initial model. More precisely, we consider the percentage of clocks merged according to the procedure described in Section 3.4.

Results: We report in Figures 10 and 11 two sets of boxplots capturing the number of states and transitions belonging to the models inferred by *TkT*. The charts show that the number of traces do not largely influence the size of the inferred automata. Indeed, except for the cases in which only 9% and 18% traces are used for the inference, the interquartile range does not largely vary in the boxplots. This suggests that *TkT* is capable of inferring complete models even when a relatively small number of traces has been used.

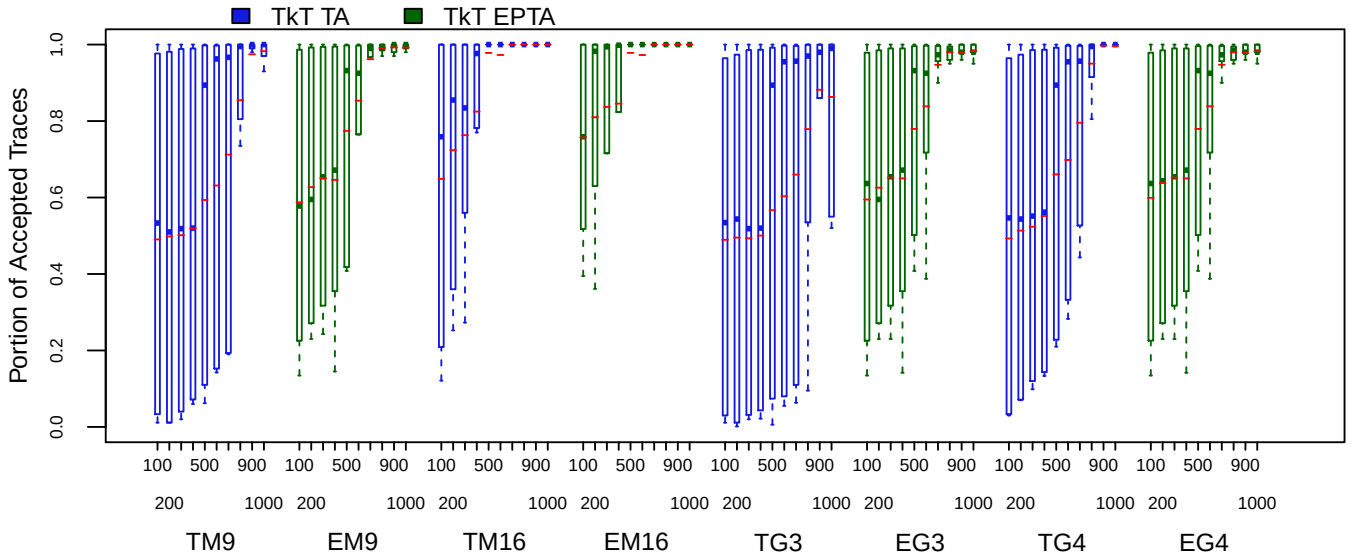
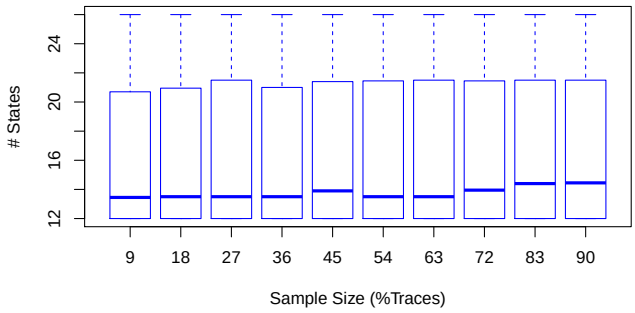
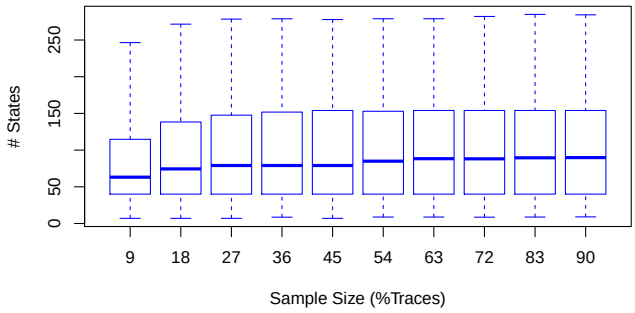


Fig. 9. RQ6: Sensitivity of *TkT* (i.e., percentage of traces that are correctly accepted) for traces of increasing length (i.e., generated by processing input collections containing 100 to 1,000 elements).

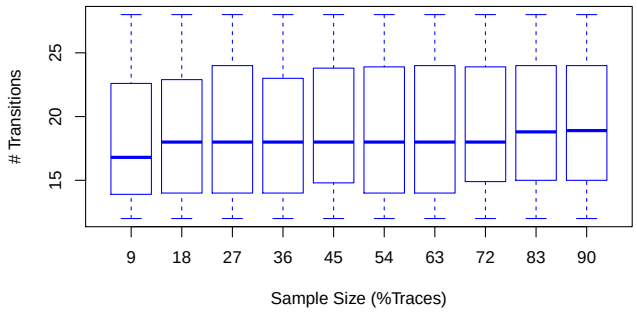


(a) Algorithms

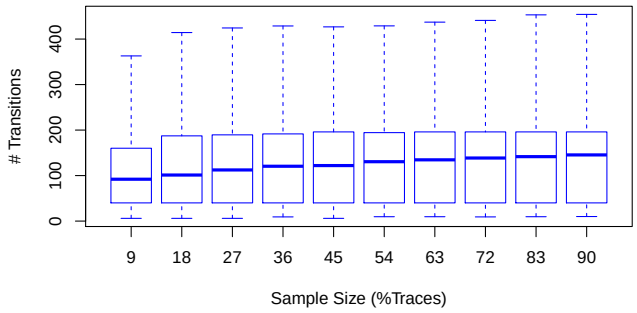


(b) Libraries

Fig. 10. RQ7: Number of states belonging to the models inferred to answer RQ1.



(a) Algorithms



(b) Libraries

Fig. 11. RQ7: Number of transitions belonging to the models inferred to answer RQ1.

Regarding clock reduction, the actual degree of reduction depends on the characteristics of both the software and the traces. However, since *TkT* assigns clocks on every begin transition, the actual reduction on the number of clocks achieved by *TkT* is proportional to the degree of generalization of the input traces. Since the inferred models are quite compact, as shown in Figures 10 and 11, also clock reduction is assumed to be significant. This is confirmed by

our experiments. Indeed, we observed an average reduction rate of the clocks comprised between 93% (for the sample size equal to 9% of the traces) and 99% (for the sample size equal to 90% of the traces) for the algorithms used in the empirical evaluation.

4.10 Threats to Validity

The choice of the faults and test cases might introduce threats to the internal validity of our study. To avoid any bias, we both distinguished the classes of faults that we investigated and we detailed the procedure that we used to work with them. In particular, we saturated the CPU to assess *TkT* against performance problems caused by anomalies in the environment, and we considered third-party confirmed performance faults to assess *TkT* against performance problems caused by software faults. In the latter case, we also defined an objective procedure based on the observed slowdown of the system to classify the test cases, and thus the relative traces, as representative of legal or illegal executions.

The external threats to validity concern with the generalization of the results. We addressed this threat by studying different classes of faults, both due to the environment and due to the executed software, different applications and several configurations, considering a range of situations, in this way mitigating the risk of lack of generalizability.

4.11 Discussion

The reported evaluation demonstrates that *TkT* can be a useful tool to derive finite state models augmented with time information that describes the behavior of a monitored program. In particular, *TkT* can generate either TAs, which use guards to represent the full duration of each operation, or EPTAs, which use guards to represent the duration of an operation without considering nested operations. Guards can be generated with two different policies, min-max ϵ policy and γ-confidence policy, both configurable according to a parameter. Finally, *TkT* can be configured to either generate or not generate constraints on the absolute clock.

The empirical results show that *TkT* is not particularly sensitive to the choice of the configuration parameters. In general, we expect developers to choose between EPTAs and TAs based on the desired semantics of the constraints. However, some software applications may have a strong cyclic behavior that cannot be correctly captured with a TA. In these cases, inferring an EPTA can be a better option, as reported in RQ6.

The choice between the two policies depends on the need of privileging sensitivity or specificity. Indeed, the γ-confidence policy performs better with specificity, while the min-max ϵ policy performs better with sensitivity. As shown, the specific choice of the values of the parameters is not fundamental, as long as small values of ϵ are avoided.

Finally, the absolute clock might improve the quality of the models, but again, if only either sensitivity or specificity should be optimized instead of the balance of the two, it might be useful to either include or exclude guards on the absolute clock to improve specificity or sensitivity, respectively.

5 Related Work

Specification mining literature includes a variety of techniques and approaches for generating models that represent the behavior of software systems from sets of execution traces. Here we discuss specification mining techniques according to the type of model that they generate: ordering of operations, simple FSAs, annotated FSAs, and FSAs with time information.

Mining Operation Order: Many techniques can generate models about the ordering of operations. Some approaches focus on partial models, that is, models that constrain the ordering of a portion of the operations that can be executed by a system. A popular way to express properties about partial order of operations is with temporal rules that specify temporal relations among events, without specifying the full behavior of the software. Various classes of temporal rules can be mined with techniques such as Perracotta [20], the algorithm by Lo et al. [44], and Texada [21].

In contrast with these approaches, which are useful when the developer is interested in properties about a subset of the events that can be produced by a software system, *TkT* aims to capture the whole behavior of a software system with a finite state model that comprehensively represents the possible combination of events that can be produced by the monitored software.

Mining Simple FSAs: Mining accurate finite state models from execution traces is a long living problem. In their seminal work, Biermann and Feldman introduced k-Tail [36], a well-known algorithm for the generation of a finite state automaton from a set of observations using an algorithm based on iterative state merging. The k-Tail algorithm has inspired many other techniques that modified and improved the learning process in different ways, for instance making the resulting model more compact [40] and introducing a steering process to improve the quality of the model [16]. Also, *TkT* uses a modified version of the state merging process originally introduced in k-Tail.

When the traces include information about the state of the monitored application, it is possible to apply a style of inference that exploits state information rather than the sequences of the events. Some of the approaches that apply this strategy are ADABU [55], ReAjax [47], and Revolution [45]. These techniques can be extremely effective, but they can be applied only when suitable monitors that can efficiently inspect the state of the application can be implemented. We designed *TkT* to be applicable to traces that are frequently available and easy to produce, such as traces with events and timestamps, for this reason *TkT* does not rely on the existence of monitors that can extract additional state information.

Mining Annotated FSAs: Few specification mining techniques deal with the problem of deriving FSAs that include additional infromation necessary to better capture the behavior of an application. This is the case of specification mining techniques that infer extended finite state machines [17], [18], [19] (i.e., FSAs with transitions annotated with guards conditions specifing the values that can be assigned to some variables), and FSA annotated with data-flow information [56] (i.e., FSAs where transitions are labeled with identifiers that can capture how the values of

some variables reoccur across events). While the annotated models can represent behavioral information that cannot be captured with simple FSAs, inferring extended models may negatively affect the quality of the models in some of the cases [48].

TkT also mines behaviors that cannot be captured with simple FSAs. Contrarily to existing extensions that focus on adding different classes of constraints on the models, *TkT* focuses on time information, which is an aspect that requires a specific extension to the mining process. The time information represented in the model produced by *TkT* requires the generation of clocks, resets, and guards on clocks, which cannot be represented with the existing extended models.

Mining FSAs with Time Information: Mining models that represent both the functional behavior of an application, such as the operations that can be performed and their ordering, and time information, such as the time required by each operation to complete, is important to be able to work on the interplay between these two aspects.

Perfume [10] is a technique that addresses the problem of mining models that capture the interplay between the functional and the timing behavior by extending Synoptic [14] with the ability to deal with the durations of the operations.

In addition to Perfume, several techniques for the inference of automata with time information [24], [25], [26], [27], [28] have been developed in recents years, but most of them have been specifically designed to model embedded and cyber-physical systems. OTALA [24] derives timed automata where each distinct combination of discrete input/output signals corresponds to a state of the automata, a criterion that cannot be easily applied to traces recorded from other software systems. BUTLA [25] instead, derives a prefix tree acceptor that captures the sequences of system events appearing in traces. BUTLA integrates a state merging criterion that merges states with a similar probability of being final, or receiving the same events. This merging criterion is effective in the context of cyber-physical systems where the system state may directly depend on individual events (e.g., a switch turned on), but it is hardly applicable to other types of software systems where the current system state may depend on sequences of operations. HyBUTLA [26] augments BUTLA with the capability of dealing with discrete and continuos signals. Contrarily to BUTLA and HyBUTLA, *TkT* targets generic software systems that can execute sequences of possibly nested operations by suitably extending the state merging strategy originally defined in k-Tail.

Several approaches, [24], [26], [27], [28] derive real-time automata [27], that is, one-clock timed automata [30] where constraints simply capture the time difference between consecutive events. *TkT* instead derives constraints that can bound the duration of pair of non-consecutive events (e.g., the beginning and the end of the execution of a method), thus being able to effectively deal with the duration of nested operations. This same limitation characterizes also approaches that derive event recording automata [29].

Finally, mining techniques can be used to associate time information with call paths [57]. The idea is to detect and analyze "hot code functions" (i.e., functions that are executed frequently) to create performance assertions, useful for performance monitoring and regression testing. In contrast with this approach, *TkT* aims to generate comprehensive finite state specifications enriched with time information.

6 Conclusions

Mining accurate models that can serve software engineering tasks is challenging. So far, specification mining solutions focused on the generation of behavioral models that capture the functional behavior of a system, producing models like temporal rules [20], [21], [44], finite state machines [16], [36], [47], and extended finite state machines [17], [18].

Although these models are useful, they fail to capture the interplay between the functional and the timing behavior of a system. This relation is relevant in many contexts where completing operations with unusual timing might represent a problem. For instance, an algorithm that requires too long to complete may cause serious inefficiencies in a system.

In this paper we presented *TkT*, a specification mining technique that generates timed automata and extended pushdown automata from a set of execution traces. The generated automata can suitably represent both the functional and the timing aspects, supporting analyses that consider those two aspects independently or jointly. Empirical results obtained with well-known data processing algorithms and leading opensource libraries has shown that *TkT* is efficient and effective. More precisely, all possible configurations of *TkT* lead to models that correctly generalize the software behavior observed in traces (i.e., with a very high sensitivity, above 0.8), even when a small portion of the available traces is used. *TkT* has high specificity, that is, it can effectively detect anomalies caused by either overloaded environment and performance faults. *TkT* has an almost optimal balance between specificity and sensitivity, independently from its configuration parameters. Interestingly, extended pushdown automata can be more effective than timed automata when the software under analysis has a strongly cyclic behavior. Finally, *TkT* scales to systems producing large execution traces.

Acknowledgments

This work 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).

References

[1] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[2] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons Inc, 2007.

[3] H. C. Gall and M. Lanza, "Software evolution: Analysis and visualization," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[4] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2001.

[5] S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: Semantic differencing for activity diagrams," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2011.

[6] F. Pastore, L. Mariani, A. Hyvarinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad, "Verification-aided regression testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[7] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 2, pp. 243–257, 2012.

[8] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[9] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Nickovic, "Automatic failure explanation in CPS models," in *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM)*, 2019.

[10] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral resource-aware model inference," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2014.

[11] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 4, pp. 486–508, 2011.

[12] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002.

[13] A. Babenko, L. Mariani, and F. Pastore, "Ava: Automated interpretation of dynamically detected anomalies," in *proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[14] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2011.

[15] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 4, pp. 408–428, 2015.

[16] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2009.

[17] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.

[18] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Journal of Empirical Software Engineering (EMSE)*, vol. 21, pp. 811–853, 2016.

[19] L. Mariani, M. Pezzè, and M. Santoro, "GK-Tail+ an efficient approach to learn software models," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 8, pp. 715–738, 2017.

[20] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[21] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015.

[22] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 99–123, 2001.

[23] K. Li, C. Reichenbach, Y. Smaragdakis, and M. Young, "Second-order constraints in dynamic invariant inference," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2013.

[24] A. Maier, "Online passive learning of timed automata for cyber-physical production systems," in *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, 2014.

[25] A. Maier, A. Vodenčarevič, O. Niggemann, R. Just, and M. Jäger, "Anomaly detection in production plants using timed automata," in *Proceedings of the International Conference on Informatics in Control Automation and Robotics (ICINCO)*, 2011.

[26] O. Niggemann, B. Stein, A. Vodencarevic, A. Maier, and H. K. Buning, "Learning behavior models for hybrid timed systems," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2012.

[27] S. Verwer, M. De Weerdt, and C. Witteveen, "An algorithm for learning real-time automata," in *Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn)*, 2007.

[28] J. Schmidt and s. Kramer, "Online induction of probabilistic real time automata," in *Proceedings of the International Conference on Data Mining (ICDM)*, 2012.

[29] O. Grinchtein, B. Jonsson, and P. Pettersson, "Inference of event-recording automata using timed decision trees," in *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, 2006.

[30] C. Martin-Vide, B. Truthe, S. Verwer, M. de Weerdt, and C. Witteveen, "The efficiency of identifying timed automata and the power of clocks," *Information and Computation*, vol. 209, no. 3, pp. 606 – 625, 2011.

[31] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[32] M. Benerecetti, S. Minopoli, and A. Peron, "Analysis of timed recursive state machines," in *Proceedings of theInternational Symposium on Temporal Representation and Reasoning (TIME)*, 2010.

[33] F. Pastore, D. Micucci, and L. Mariani, "Timed k-tail: Automatic inference of timed automata," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[34] Apache Software Foundation, "Commons collections library," https://commons.apache.org/proper/commons-collections/.

[35] ——, "Commons math library," http://commons.apache.org/proper/commons-math/.

[36] A. Biermann and J. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 21, no. 6, pp. 592 – 597, 1972.

[37] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proceedings of the Workshop on Verification and Control of Hybrid Systems III*, 1995.

[38] Z. Dang, "Pushdown timed automata: a binary reachability characterization and safety verification," *Theoretical Computer Science*, vol. 302, no. 1, pp. 93 – 121, 2003.

[39] F. Pastore, L. Mariani, and D. Micucci, "BDCI: Behavioral driven conflict identification," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2017.

[40] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, pp. 215–249, 1998.

[41] C. Daws and S. Yovine, "Reducing the number of clock variables of timed automata," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[42] F. Dekking, C. Kraaikamp, H. P. Lopuhaa, and L. E. Meester, *A Modern Introduction to Probability and Statistics*. Springer, 2005.

[43] A. V. Raman, J. D. Patrick, and P. North, "The sk-strings method for inferring pfsa," in *Proceedings of the Workshop on automata induction, grammatical inference and language acquisition at the international conference on machine learning (ICML)*, 1997.

[44] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," *Science of Computer Programming*, vol. 77, no. 6, pp. 743–759, 2012.

[45] L. Mariani, A. Marchetto, C. Nguyen, P. Tonella, and A. Baars, "Revolution: Automatic evolution of mined specifications," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2012.

[46] C. Boufaied, D. Bianculli, and L. C. Briand, "A model-driven approach to trace checking of temporal properties with aggregations," *Journal of Object Technology (JOT)*, vol. 18, no. 2, pp. 1–21, 2019.

[47] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, 2008.

[48] D. Lo, L. Mariani, and M. Santoro, "Learning extended fsa from software: An empirical assessment," *Journal of Systems and Software (JSS)*, vol. 85, no. 9, pp. 2063 – 2076, 2012.

[49] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[50] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[51] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[52] Marc R. Hoffmann and Brock Janiczak and Evgeny Mandrikov, "Eclemma," https://www.eclemma.org/, 2020.

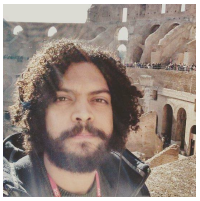
[53] Google, "Java collections library," https://github.com/google/guava.
[54] Eclipse, "Aspectj," https://eclipse.org/aspectj/, 2020.
[55] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *Proceedings of the International Workshop on Dynamic Analysis (WODA)*, 2006.
[56] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2008.
[57] M. Brünink and D. S. Rosenblum, "Mining performance specifications," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

Fabrizio Pastore is Chief Scientist II at the SnT Centre for Security, Reliability, and Trust of the University of Luxembourg. He obtained a PhD in Computer Science in 2010 from the University of Milano - Bicocca.

His research interests concern specifications mining and automated software testing, including security testing, based on different types of data and analyses (models, requirements, program analysis, crowdsourcing). He is active on EU-funded research projects and several industry partnerships.

Daniela Micucci obtained her Ph.D. in Mathematics, Statistics, Computational Sciences and Computer Science from the University of Milano in 2004. She is currently an assistant professor at the University of Milano Bicocca.

Her research interests include software engineering, in particular software architectures, real-time systems, and self-healing and self-repairing systems. She is currently active in several European and National projects. She is also regularly involved in the program committees of workshops and conferences in her areas of interest.



Michell Guzman is a postdoctoral researcher at University of Milano-Bicocca. He obtained his PhD in Computer Science in 2017 from l'École Polytechnique de Paris.

His research interest include formal verification, modal and epistemic logics, concurrency theory, process calculi, distributed computing, constraint systems, bioinformatics, self-healing and self-repairing systems. He is currently working in self-healing and policy enforcement at the Department of Informatics, Systems and Communication of the University of Milano-Bicocca.

Leonardo Mariani is a Full Professor at the University of Milano Bicocca. He holds a Ph.D. in Computer Science received from the same university in 2005.

His research interests include software engineering, in particular software testing, program analysis, automated debugging, specification mining, and self-healing and self-repairing systems. He has authored more than 100 papers appeared at top software engineering conferences and journals.

He has been awarded with the ERC Consolidator Grant in 2015, an ERC Proof of Concept grant in 2018, and he is currently active in several European and National projects. He is regularly involved in organizing and program committees of major software engineering conferences.