CkTail: Model Learning of Communicating Systems

Sébastien Salva and Elliott Blot
LIMOS - UMR CNRS 6158, Clermont Auvergne University, France

Keywords: Reverse Engineering, Model Learning, Communicating Systems, IOLTS, Dependency Graphs.

Abstract: Event logs are helpful to figure out what is happening in a system or to diagnose the causes that led to an unexpected crash or security issue. Unfortunately, their growing sizes and lacks of abstraction make them difficult to interpret, especially when a system integrates several communicating components. This paper proposes to learn models of communicating systems, e.g., Web service compositions, distributed applications, or IoT systems, from their event logs in order to help engineers understand how they are functioning and diagnose them. Our approach, called CkTail, generates one Input Output Labelled Transition System (IOLTS) for every component participating in the communications and dependency graphs illustrating another viewpoint of the system architecture. Compared to other model learning approaches, CkTail improves the precision of the generated models by better recognising sessions in event logs. Experimental results obtained from 9 case studies show the effectiveness of CkTail to recover accurate and general models along with component dependency graphs.

1 INTRODUCTION

Using event logs to debug systems in the short or long term is an approach more and more considered in the Industry. Logs strongly help investigate issues on production environments, and usually increase the developers' ability to handle and detect failures. But, it is well-known that analysing log entries is often a long and frustrating process as it usually requires to cover very large files. Several approaches based upon model learning propose to ease log analysis by building models that generalise the behaviours of systems encoded in logs and by making them readable (Krka et al., 2010; Ohmann et al., 2014; Salva and Blot, 2019; Salva and Durand, 2015; Pastore et al., 2017; Biermann and Feldman, 1972; Mariani and Pastore, 2008; Beschastnikh et al., 2014). We consider in the paper passive model learning approaches *that infer a specification by gathering and analysing system executions and concisely summarising the frequent interaction patterns as state machines that capture the system behaviour* (Ammons et al., 2002). The obtained models, even if partial, can serve many purposes, e.g., they can be used as documentation, examined by designers to find bugs, or can be given to testing methods.

This paper focuses on the generation of models of communicating systems from event logs although some works already proposed solutions (Mariani and Pastore, 2008; Beschastnikh et al., 2014; Salva and Blot, 2019). The algorithm given in (Mariani and Pastore, 2008) segments event logs into execution traces by considering two factors, component identification and user identification, then models are built using the kBehavior algorithm (Mariani et al., 2011). The CSight tool (Beschastnikh et al., 2014) takes as inputs trace sets, which have to be manually built by hands w.r.t. to several restrictions. Models are then generated and refined by means of invariants. We also proposed the tool Assess (Salva and Blot, 2019), which is specialised in the model learning of component based systems from which the component interactions are not observable. Assess splits event logs into traces by looking for longer time delays between messages. Then, it tries to detect implicit component calls in traces and builds models encoding these calls with new synchronisation actions. We observed that these previous approaches suffer from one main issue related to session recovery in event logs. We call a session a temporary message interchange among components forming a behaviour of the whole system from one of its initial states to one of its final states. Recognising sessions in event logs helps extract "complete" traces and build more precise models. It is quite straightforward to recognise sessions when a mechanism based on session identification is used. Unfortunately, many kinds of systems do not use such mechanisms.

Salva, S. and Blot, E.
CkTail: Model Learning of Communicating Systems.
DOI: 10.5220/0009327400270038
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 27-38
ISBN: 978-989-758-421-3
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Our approach, called Communicating system kTail shortened CkTail, builds more precise models of communicating systems by better recognising sessions in event logs with respect to 4 properties: association of responses with their related requests, time delays, data dependency and component identification. To design CkTail, we choose to extend the k-Tail technique (Biermann and Feldman, 1972) with the capability to build one model called Input Output Labelled Transition System (IOLTS) for every component of the system under learning. k-Tail is indeed well-know to quickly and efficiently build generalised models from traces. Furthermore, our approach also goes further in model learning by proposing the generation of dependency graphs. The latter show in a simple way the directional dependencies observed among components. We believe that this kind of graph completes the behavioural models and will be helpful to evaluate different kinds of model properties, e.g., testability or security. In the paper, we define component dependency over three expressions formulating these situations: direct component requests, nested requests and data dependency. This definition aims to avoid the inference of ambiguous dependency relations involving one component to several potential components.

We performed an empirical evaluation based on event logs collected from 9 case studies to assess the precision of the models and of the dependency graphs. We show that CkTail is more effective than the three previously cited approaches.

The paper is organized as follows: we recall some definitions about the IOLTS model in Section 2. Our approach is presented in Section 3 with a motivating example. The next section shows some results of our experimentations. Section 5 discusses related work. Section 6 summarises our contributions and draws some perspectives for future work.

2 THE IOLTS MODEL

We express the behaviours of communicating components with IOLTSs. This model is defined in terms of states and transitions labelled by input or output actions, taken from a general action set $\mathcal{L}$, which expresses what happens. τ is a special symbol encoding an internal (silent) action; it is common to denote the set $\mathcal{L} \cup \tau$ by $\mathcal{L}_\tau$.

Definition 1 (IOLTS). *An Input Output Labelled Transition System (IOLTS) is a 4-tuple $\langle Q, q0, \Sigma, \rightarrow \rangle$ where:*

- *Q is a finite set of states; $q0$ is the initial state;*
- *$\Sigma \cup \{\tau\} \subseteq \mathcal{L}_\tau$ is the finite set of actions, with τ the internal (unobservable) action. $\Sigma_I \subseteq \Sigma$ is the countable set of input actions, $\Sigma_O \subseteq \Sigma$ is the countable set of output actions, with $\Sigma_O \cap \Sigma_I = \emptyset$;*
- *$\rightarrow \subseteq Q \times \Sigma \cup \{\tau\} \times Q$ is a finite set of transitions. A transition (q, a, q') is also denoted $q \xrightarrow{a} q'$.*

A trace is a finite sequence of observable actions in $\mathcal{L}^*$. For sake of readability, we also write $l \in l'$ when l is a subsequence of the sequence l'; $last(l)$ refers to the last element of l. Furthermore, to better match the functioning of communicating systems, we assume that an action has the form $a(\alpha)$ with a a label and α an assignment of parameters in P, with P the set of parameter assignments. For example, $switch(from := c_1, to := c_2, cmd := On)$ is made up of the label "switch" followed by a parameter assignment expressing the components involved in the communication and a parameter of the switch command.

3 THE CkTail APPROACH

Given an event log, collected by monitoring tools on a system under learning denoted SUL, CkTail extracts traces, generates one IOLTS for every component or software of SUL involved in the communications and builds its dependency graph. Here, an IOLTS aims at generalising the behaviours encoded in event logs with inputs and outputs showing the messages received and sent by a component.

The ability of CkTail to infer models is dependent on several realistic assumptions made on SUL:

- **A1 Event Log:** the communications among the components can be monitored by different techniques, on components, on servers, by means of wireless sniffers, etc. But, event logs have to be collected in a synchronous environment made up of synchronous communications. Furthermore, the messages have to include timestamps given by a global clock for ordering them. For simplicity, we consider having one event log at the end of the monitoring process;
- **A2 Message Content:** components produce messages that include two parameter assignments of the form $from := c$, $to := c$ expressing the component source and the destination of a message. Other parameter assignments may be used to encode data. Besides, a message is either identified as a request or a response. Many protocols allow to easily and automatically distinguish both of them;

- **A3 Device Collaboration:** components can run in parallel and communicate with each other. To recognise sessions of the system in event logs, we assume that components follow this behaviour: they cannot run multiple instances; requests are processed by a component on a first-come, first served basis. Besides, every response is associated to a request and vice-versa.

The last assumption helps process event logs with the aim of extracting the messages of each component into disjointed sequences of individual sessions. It is possible to replace this assumption by the more classical one stating that messages include an identifier allowing to observe whole collaborations among components. This last assumption strongly eases the trace extraction from event logs. Unfortunately, we have observed that session identification is seldom used with communicating systems. Instead, we have chosen to consider the assumption A3. But it is worth noting that if a session mechanism is used in SUL, the first step of CkTail may be adapted (simplified) to still detect dependencies among components.

3.1 CkTail Overview

Figure 1 illustrates an overview of the three steps of CkTail. Initially, the user gives as inputs an event log along with regular expressions. The raw messages of the event log are firstly parsed and analysed with these expressions to structure them into actions of the form $a(\alpha)$ with a a label and α some parameter assignments. Figure 1 shows a list of 12 actions derived from raw HTTP messages. The action structure meets the previous assumptions: the assignments of the variables $from$ and to indicate the sources and destinations of the messages. For sake of readability, the labels directly show whether an action encodes either a request or a response. In this example, the other parameter assignments are data expressing commands (e.g., param:=heating, cmd:=on), temperature values (e.g., param:=udevice, svalue:=66) or component states (e.g., param:=udevice, svalue:=open). At this stage, we can observe that there are 5 components. But interpreting their interactions and what they can do is still tricky because of lack of readability and generalisation.

The first step of CkTail is called *Trace analysis*. It covers the action list derived from the event log and aims at segmenting it into traces that capture sessions. A trace is a subsequence of the action list that meets some constraints formulated from the assumptions A1-A3. These constraints are detailed in Section 3.2 and summarised as follows. A response is always associated to its related request in a trace (A3). Nested requests (a request to a component that also performs another request before giving a response) are always kept together in a trace (A3). A trace gathers messages exchanged between components interacting together in a limited time delay (A1). And, a chain of messages sharing the same data expresses a data dependency among several components. This chain of messages must be kept in the same trace (A2).

During this process, the component interactions are also analysed for detecting component dependencies. These dependencies are given under the form of component lists $c_1 c_2 \dots c_k$ expressing that a component c_1 depends on a component c_2, which itself depends on another component and so on. The set $Deps$ gathers these component lists. The example of set $Deps$ of Figure 1 captures several dependencies. For instance, $d_1 G$ is given by $req1$ as c_1 directly calls G. $d_1 G d_2$ is given from the nested requests req1req3, showing that d_1 calls G, which itself calls d_2 before answering to d_1. $d_2 d_3$ is a data dependency observed between d_2 and d_3 related to the shared value (svalue:=68) exchanged between the two components.

The second step *Dependency Graph Generation* takes back the set $Deps$, constructs Direct Acyclic Graphs (DAGs) from the component lists and eventually computes the transitive closure of the DAGs. These DAGs only capture the dependencies given in $Deps$, i.e. two different lists $c_1 c_2$ and $c_2 c_3$ are not associated to avoid the representation of false links among components. The dependency graphs illustrated in Figure 1 (top right) show the dependent components of every component of SUL. The components required by some other ones can be retrieved though. These graphs also show in our example that the component G (a gateway) takes a central place as it is required for d1 and d3, and it depends on d2 and d4.

The third step *Model Generation* builds IOLTSs. In the figure, the *Step 3A Trace Partitioning* begins by preparing the set $Traces(\text{SUL})$. Every action is doubled to give a pair output/input by separating the notion of source/destination by means of the new parameter *id* assigned either to the source or to the destination of a message. During this process, $Traces(\text{SUL})$ is partitioned into as many trace sets as components found in SUL. Each trace set T_c gathers only the traces related to the component c. The *Step 3B IOLTS Generation* transforms every set T_c into an IOLTS by converting traces into IOLTS path cycles, which are joined on the initial state only. In our example, as we have 5 trace sets, we obtain 5 IOLTSs. Finally, the *Step 3C IOLTS Reduction*, applies the k-Tail algorithm to the IOLTSs in order to merge their equiv-

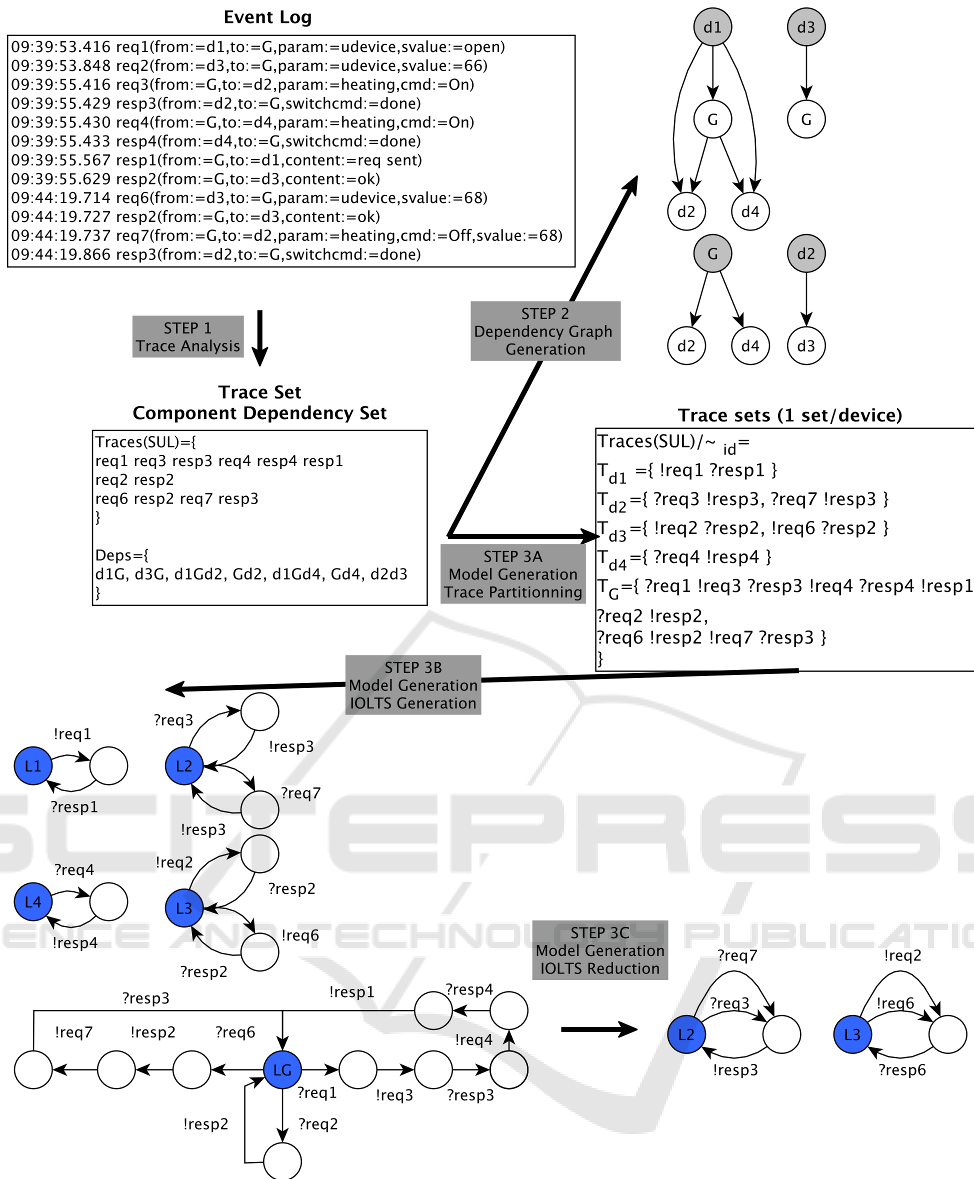

Figure 1: Approach overview.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

alent states. With k-Tail, the equivalent states are those having the same k-future, i.e. the same event sequences having the maximum length k. In our example, the states in white of the two IOLTSs L_2 and L_3 are merged.

With these IOLTSs and the DAGs, it becomes easier to observe that SUL is made up of two sensors d_1 and d_3 connected to a gateway G. The first is a motion detector providing states and the second gives temperature values. The gateway controls two actuators, here two heating systems d_2, d_4, with respect to the values provided by the sensors. When the d_1 state is on, the heating systems are turned on. When d_3 sends a temperature upper to 68, the gateway turns off the heating system d_2 only and forwards the temperature to d_2 in the meantime.

The CkTail steps are now detailed in the next sections.

3.2 Trace Analysis

As stated previously, this first step covers an event log to extract the trace set $Traces(\text{SUL})$ and the set $Deps$ of dependency lists. The messages of the event log are firstly parsed and analysed with regular expressions to retrieve actions of the form $a(\alpha)$. If the user is not able to give these regular expressions, several approaches and tools have been proposed to automat-

ically mine patterns from log files (Fu et al., 2009; Makanju et al., 2012; Vaarandi and Pihelgas, 2015; Messaoudi et al., 2018; Zhu et al., 2018). These patterns may be used to quickly derive regular expressions. The resulting actions have to meet the requirements given by the assumptions A1-A3. We define the following notations to express some of these requirements:

Definition 2. *Let $a(\alpha)$ be an action in $\mathcal{L}_\tau$.*

- $from(a(\alpha)) = c$ *denotes the source of the message,*
- $to(a(\alpha)) = c$ *denotes the destination,*
- $components(a(\alpha)) = \{from(a(\alpha)), to(a(\alpha))\}$,
- $time(a(\alpha))$ *denotes the timestamps,*
- $data(a(\alpha)) = \alpha \setminus components(a(\alpha)) \setminus time(a(\alpha))$,
- $isReq(a(\alpha))$, $isResp(a(\alpha))$ *are boolean expressions expressing the nature of the message.*

Once we have the list of actions, which we denote S, this step segments it into traces by trying to recover sessions. As we assume the latter are not identified, we propose an algorithm that detects them by following the assumptions A1-A3. To devise this algorithm, we thoroughly derived a complete list of constraints from A1-A3. Some of them are given in Table 1. Given an action $a(\alpha)$, some constraints forbids starting a new trace from $a(\alpha)$: with C1, a response must be kept with its associated request in the same trace; similarly, with C2, nested requests must remain together in the same trace. On the contrary, other constraints imply to cut the action list. For instance, C3 implies that a new component not yet observed that sends a new request, is starting a new session. We hence consider that this request starts a new trace. Other constraints, e.g., C4, reflect incoherent cases that do not meet our assumptions. C5 is a special constraint expressing that a component, which has previously participated in the current session, can send a new request. The choice of keeping this new request or not in the current session is not obvious. To make this choice, we propose to consider two other factors, i.e., time delay and data dependency, with the constraints C5.1 and C5.2. The notion of data dependency will be presented in Section 3.3.

To be used by our trace analysis algorithm, we have formulated these constraints with the notations of Definition 2 completed with the following sets and boolean expressions. We denote $Lreq$ the set of lists of pending requests made in parallel at a given time, i.e. lists of actions $a_1(\alpha_1)\dots a_k(\alpha_k)$ with $isReq(a_i(\alpha_i))_{1\leq i\leq k}$ for which responses have not yet been received. KC denotes the list of

Table 1: Some constraints derived from the assumptions A1-A3.

C1	A response $a_i(\alpha_i)$ is associated to a request observed previously in the same session
C2	A component that receives a request $a_i(\alpha_i)$ can call other components to produce a response in the same session.
C3	A component not yet encountered that has no pending request can send a new request $a_i(\alpha_i)$ and starts a new session.
C4	A component that previously requested another component, cannot send a new request $a_i(\alpha_i)$, it needs to wait for the response of the first one.
C5	A component that has completed its request flow (received or sent requests) in a session, can send a new request $a_i(\alpha_i)$ to another component. C5.1: if the time delay between this request and the previous request made by this component is short, the request belongs to the session. C5.2: if this request shares data with previous actions, then the request belongs to the same session as these actions.

Table 2: Constraint formalisation. C1 C2 and C5 are the constraints allowing an action to be kept in a current trace.

	Constraint
C1	$isResp(a_i(\alpha_i))$ and $\exists! l \in Lreq$: $from(a_i(\alpha_i)) = to(last(l))$ and $to(a_i(\alpha_i)) = from(last(l))$
C2	$isReq(a_i(\alpha_i))$ and $Lreq' = \{l \in Lreq \mid from(a_i(\alpha_i)) = to(last(l))\} \neq \emptyset$ and $\neg pendingRequest(from(a_i(\alpha_i)))$
C5	$isReq(a_i(\alpha_i))$ and $from(a_i(\alpha_i)) \in$ KC and $\forall l \in Lreq$: $from(a_i(\alpha_i)) \neq to(last(l))$ and $\neg pendingRequest(from(a_i(\alpha_i)))$

known components involved in the session so far. $pendingRequest(c)$ is the boolean expression ($\exists c \in KC, \exists l \in Lreq, a(\alpha) \in l : c \in components(a(\alpha))$) that evaluates whether the component c has sent (resp. received) a request and has not yet received (resp. sent) the response. From these notations, we have formulated the above constraints, listed their boolean terms and studied all their possible permutations. Table 2 lists the constraints expressing that an action $a_i(\alpha_i)$ must be kept in the current trace when they hold.

Algorithms 1 and 2 implement the above constraints and segment an action list S into traces. This recursive algorithm is initially called with $KeepSplit(S,S)$; it returns $Traces(\mathrm{SUL})$ and the final component set C, which differs from KC, the set of known components in a session. If the action list S starts with a response (without having any request before), it is deleted (lines 1-3). Otherwise, a new trace t is started with a first request. The list of pending requests $Lreq$ is updated with the procedure $Extend$ and the set of known components KC is completed w.r.t. this request. Then, every action $a_i(\alpha_i)$ of the list S is covered to decide whether the action is kept in the current trace (line 9). If the constraint C1 holds (receipt of a response associated to a previous request), $a_i(\alpha_i)$ is added to the trace t. The associated pending request is removed from $Lreq$ with the procedure $Trim$. If C2

holds (receipt of a request that is nested to a previous request in *Lreq*), $a_i(\alpha_i)$ is also added to the trace t. To check whether C2 holds, a subset $Lreq'$ of *Lreq* is constructed: it is composed of the request sequences ended by a request $a(\alpha)$ such that $a(\alpha)a_i(\alpha_i)$ gives form to nested requests. If $Lreq'$ has several candidates, i.e. request lists, we keep the list l ended by the request having the earliest time-stamp (line 13). With this condition, we comply with A3 (first-come, first served basis). This list l is augmented with the current request $a_i(\alpha_i)$ by calling the procedure *Extend* (line 14). The set of known components *KC* is updated. Finally, when C5 holds (line 15), we evaluate the constraints C5.1 or C5.2 implemented by the procedures *Checktime* and *Checkdatadeps*. If one of these procedures returns true, the request $a_i(\alpha_i)$ is added to the trace t. The procedure *Extend* completes *Lreq* with a new request list starting by $a_i(\alpha_i)$. The set *KC* is still updated w.r.t. the request $a_i(\alpha_i)$. For any other case, we put the action $a_i(\alpha_i)$ into a new action list t_2 (line 18). Once all the actions have been covered, the trace t is added to $Traces(\text{SUL})$. If the action list t_2 is not empty, the algorithm is called again to split it (line 23).

The procedure *Checktime* takes a trace t and a request $a_i(\alpha_i)$. Its purpose is to check whether the time delay between this request and the previous one made by the same component c is strongly lower than the average time delay between two requests made by c. This average time delay is denoted $T(c)$. Its computation is made on the action list S. In short, we filter out all the responses along with the requests not made by the component c in S. Then, we compute the mean of time delays between every request pairs. *Checktime* also filters the trace t to keep the requests made by the component $c = from(a_i(\alpha_i))$. The procedure computes the time delay between the request $a_i(\alpha_i)$ and the previous one in the trace t. If this time delay is lower to $T(c)$, C5.1 holds, hence the procedure returns true.

The procedure *Checkdatadeps* implements the constraint C5.2 and checks whether a data dependency exists in the trace t. The notion of dependency among components is discussed in the next section.

3.3 Dependency Graph Generation

Before generating dependency graphs, we need to express what a component dependency is. Definition 3 relies on three expressions to formulate this notion. The first illustrates that a component c_1 depends on another component c_2 when c_1 queries c_2 with a request. The second expression deals with nested requests: if we have successive nested requests of the form $req1(from := c_1, to := c)req2(from := c, to := c_2)$, we define that c_1 depends on c, which itself depends on c_2 and so on. The last expression refers to data dependency. We say that c_1 depends on c_2 when c_2 has sent a message $a_1(\alpha_1)$ with some data, if there is a unique chain of messages $a_1(\alpha_1)\dots a_k(\alpha_k)$ from c_2 sharing this data and if $a_k(\alpha_k)$ is a request whose destination is c_1.

Some dependency cases, or patterns, might have been forgotten in this definition. For instance, when there are two or more chains of messages sharing the same data, addressed to the same component c, we observe that there is a data dependency among components but we are unable to decide which dependency relation is correct. Because of this ambiguity, none of them is kept.

Algorithm 1: $KeepSplit(S, S_2)$.

input : Action sequences S, S_2
output: $Traces(\text{SUL})$, Component set C

1 $S_2 = a_1(\alpha_1)\dots a_k(\alpha_k)$;
2 **while** $isResp(a_1(\alpha_1))$ **do**
3     $KeepSplit(S, a_2(\alpha_2)\dots a_k(\alpha_k))$;
4   END;
5 $t = a_1(\alpha_1)$; $Extend(Lreq, \varepsilon, t)$;
6 $t_2 = \varepsilon$;
7 $\text{KC} = components(a_1(\alpha_1))$;
8 i=2;
9 **while** $i \leq k$ **do**
10   Case C1 : $t = t.a_i(\alpha_i)$; $Trim(Lreq, l)$;
11   Case C2 :
12         $t = t.a_i(\alpha_i)$;
13   take $l \in Lreq'$ such that $\forall l_2 \neq l \in Lreq' : time(last(l)) \leq time(last(l_2))$;
14         $Extend(Lreq, l, t)$;
15   Case C5 and ($Checktime(a_i(\alpha_i), t)$ or $Checkdatadeps(a_i(\alpha_i), S, t)$) :
16         $t = t.a_i(\alpha_i)$;
17         $Extend(Lreq, \varepsilon, t)$;
18   Else : $t_2 = t_2.a_i(\alpha_i)$;
19     $\text{KC} = \text{KC} \cup components(a_i(\alpha_i))$;
20   i++;
21 $Traces(\text{SUL}) = Traces(\text{SUL}) \cup \{t\}$;
22 $C = C \cup \text{KC}$;
23 **if** $t_2 \neq \varepsilon$ **then**
24     $KeepSplit(S, t_2)$;

Definition 3 (Component Dependency). *Let $c_1, c_2 \in C$, $c_1 \neq c_2$, and S an action list. We write c_1 depends on c_2 iff:*

1. *c_1 calls c_2 : $\exists a(\alpha) \in S : isReq(a(\alpha))$, $from(a(\alpha)) = c_1, to(a(\alpha)) = c_2$;*
2. *there is a nested request between c_1 and c_2 : $\exists a_1(\alpha_1), a_2(\alpha_2) \in S$, $\forall a(\alpha) \in S$: $isReq(a_1(\alpha_1))_{(i=1,2)}$, $isResp(a(\alpha))$, $from(a_1(\alpha_1)) = c_1$, $to(a_1(\alpha_1)) = from(a_2(\alpha_2))$,*

$to(a_2(\alpha_2)) = c_2$, $to(a(\alpha)) = c_1$, $\neg(a_1(\alpha_1)a(\alpha)a_2(\alpha_2) \in S)$;

3. *there is a longest chain of messages from* c_2 *ended by a request to* c_1 *sharing the same data* α: $\exists l = a_1(\alpha_1)a_2(\alpha_2)\ldots a_k(\alpha_k) \in S : DS(l, c_1, c_2, \alpha)$ *and* $\forall l' = a'_1(\alpha'_1)a'_2(\alpha'_2)\ldots a_k(\alpha_k) \in S : DS(l', c_1, c_2, \alpha)$ *and* $l' \in l$, *with* $DS(a_1(\alpha_1)\ldots a_k(\alpha_k)c_1, c_2, \alpha)$ *the boolean expression* $from(a_1(\alpha_1)) = c_2 \wedge to(a_k(\alpha_k)) = c_1 \wedge isReq(a_k(\alpha_k)) \wedge\ to(a_i(\alpha_i)) = from(a_{i+1}(\alpha_{i+1}))_{1 \le i < k}) \wedge \alpha \subseteq \bigcap_{(1 \le i \le k)} \alpha_i$.

The set *Deps* gathers component dependencies under the form of component lists $c_1 \ldots c_k$. Component dependencies are detected while Algorithm 1 builds traces by means of the procedures *Extend* and *Checkdatadeps*. The procedure *Extend* detects the two first component dependency cases of Definition 3. It uses the set of pending requests *Lreq* to complete the set *Deps*. Indeed, Algorithm 1 builds *Lreq* in such a way that a sequence of *Lreq* is either a request (Case C5) or a list of nested pending requests (Case C2). The procedure covers the component sequences $lc = c_1 \ldots c_k c_{k+1}$ of *Lreq* and adds the dependency lists in *Deps* (line 8).

The procedure $Checkdatadeps(a_i(\alpha_i), S, t)$ checks whether the last expression of Definition 3 holds. If there is a unique chain of messages $a_1(\alpha_1)\ldots(a_i(\alpha_i)$ sharing the same data $\alpha \in data(a_i(\alpha_i))$ and finished by the request $a_i(\alpha_i)$ then the dependency $to(a_i(\alpha_i)).from(a_1(\alpha_1))$ is added to *Deps* (line 15). If this chain of messages is a subsequence of the current trace $t.a_i(\alpha_i)$, then the constraint C5.2 is satisfied. As a consequence, the procedure also returns true to Algorithm 1 to indicate that this request must be kept in the current trace.

Algorithm 3 can now generate dependency graphs from the set *Deps*. It partitions *Deps* to group the dependency lists starting by the same component in the same subset. This partitioning is performed by the equivalence relation $\sim_c$ on C^* given by $\forall l_1, l_2 \in Deps$, with $l_1 = c_1 \ldots c_k$, $l_2 = c'_1 \ldots c'_k$, $l_1 \sim_c l_2$ iff $c_1 = c'_1$. Given a partition C_i, and a component list $l \in C_i$, Algorithm 3 builds a path of the DAG Dg_i such that the nth state is labelled by the nth component of l. Algorithm 3 finally computes the transitive closure of the DAGs to make all component dependencies visible.

3.4 Model Generation

This last step, implemented by Algorithm 4, generates an IOLTS for every component previously encountered and stored in C. To build IOLTSs, the traces of $Traces(\text{SUL})$ are transformed to integrate the notions of input and output. Given a trace $a_1(\alpha_1)\ldots a_k(\alpha_k)$, every action is doubled by separating the component source and the destination of the message. For an action $a_i(\alpha_i)$, the algorithm writes a new trace composed of the output $!a_i(\alpha_{i1})$ sent by the source of the message, followed by the input $?a_i(\alpha_{i2})$ received by the destination (line 3). The source and the destination are now separated by a new assignment on the parameter *id*. Next, these traces are segmented into subsequences in such a way that a subsequence captures the behaviour of one component only. Subsequences are also grouped by component in trace sets denoted T_c with c a component of C (lines 4,5).

These trace sets are now lifted to the level of IOLTSs. Given the trace set T_c, a trace $t = a_1(\alpha_1)\ldots a_k(\alpha_k) \in T$ is transformed into the path

Algorithm 2.

```
1  Procedure Trim(Lreq, l) is
2      l' = remove(last(l));
3      Lreq = Lreq \ {l} ∪ {l'};
4  Procedure Extend(Lreq, l, t) is
5      l' = l.last(t) = a1(α1)...ak(αk);
6      Lreq = Lreq \ {l} ∪ {l'};
7      //Compute component dependencies
8      lc = c1...ck ck+1 such that ci = from(ai(αi))(1≤j≤k),
         ck+1 = to(ak(αk));
9      Deps = Deps ∪ {lc};
10 Procedure Checktime(ai(αi), t) is
11     t2 = t\{a(α) | from(ai(αi)) ≠ from(a(α)) or
         isResp(a(α))};
12     return
         (time(ai(αi)) − time(last(t2))) << T(from(ai(αi))));
13 Procedure Checkdatadeps(ai(αi), S, t) is
14     if ∃α ∈ data(ai(αi)), ∃l = a1(α1)a2(α2)...ai(αi) ∈ S :
         DS(l, to(ai(αi)), from(a1(α1)), α) and
         ∀l' = a'1(α'1)a'2(α'2)...ai(αi) ∈ S :
         DS(l', to(ai(αi)), from(a1(α1)), α) and l' ∈ l then
15         Deps = Deps ∪ {to(ai(αi)).from(a1(α1))};
16         if l ∈ t.ai(αi)) then
17             return true;
18     else
19         return false;
```

Algorithm 3: Device Dependency Graphs Generation.

```
   input : Deps
   output: Dependency graph set DG
1  foreach Ci ∈ Deps/ ∼c do
2      foreach c1c2...ck ∈ Ci do
3          add the path sc1 → sc2 ...sck−1 → sck to Dgi;
4      Dg'i is the transitive closure of Dgi;
5      DG = DG ∪ {Dg'i};
```

$q0 \xrightarrow{a_1(\alpha_1)} q_1 \dots q_{k-1} \xrightarrow{(a_k(\alpha_k)} q0$ such that the states $q_1 \dots q_{k-1}$ are new states. These paths are joined on the state $q0$ to build the IOLTS L_c:

Definition 4 (IOLTS Generation). *Let $T_c = \{t_1, \dots, t_n\}$ be a trace set. $L_c = \langle Q, q0, \Sigma, \rightarrow \rangle$ is the IOLTS derived from T_c where:*

- *$q0$ is the initial state.*
- *$Q, \Sigma, \rightarrow$ are defined by the following rule:*

$$\frac{t_i = a_1(\alpha_1)\dots a_k(\alpha_k)}{q0 \xrightarrow{a_1(\alpha_1)} q_{i1} \dots q_{ik-1} \xrightarrow{a_k(\alpha_k)} q0}$$

Finally, Algorithm 4 applies the k-Tail algorithm to generalise and reduce the IOLTSs by merging the equivalent states having the same k-future. We use $k = 2$ as recommended in (Lorenzoli et al., 2008; Lo et al., 2012).

Algorithm 4: IOLTS Generation.

```
  input : Traces(SUL)
  output: LTSs L_{c_1} ... L_{c_k}
1 T = {};
2 foreach t = a_1(α_1)...a_k(α_k) ∈ Traces(SUL)) do
3     t' = !a_1(α_11)?a_1(α_12)...!a_k(α_k1)?a_k(α_k2) such that
        α_i1 = α_i ∪ {id := from(a_i(α_i))},
        α_i2 = α_i ∪ {id := to(a_i(α_i))}(1 ≤ i ≤ k);
4     foreach c ∈ C do
5         T_c = T_c ∪ {t' \ {a(α) ∈ t' | id := c ∉ α}
6 foreach T_c with c ∈ C do
7     Generate the LTS L_c from T_c;
8     Merge the equivalent states of L_c with k-Tail(k = 2, L_c);
```

4 PRELIMINARY EVALUATION

Our approach is implemented in Java and is released as open source at https://github.com/Elblot/CkTail. With this implementation, we conducted some experiments in order to provide answers for the following questions:

- RQ1: Can CkTail derive models that capture correct behaviours of SUL? RQ1 studies how the models accept valid traces, i.e. traces extracted from event logs but not used for the model generation, compared to the model learning tools specialised for communicating systems;
- RQ2: Can CkTail build models that reject abnormal behaviours? This time, RQ2 investigates how the models accept invalid traces;
- RQ3: Is CkTail able to detect accurate dependencies among components?

4.1 Empirical Setup

RQ1 and RQ2 evaluate the precision of the models generated by CkTail. We here compare CkTail with 3 other tools, CSight, Assess and the tool suite proposed in (Mariani and Pastore, 2008) based upon the tool kbehavior, which we denote LFkbehavior. To perform an unbiased evaluation, we need to take into consideration how these tools work. Assess is specialised to the model learning of component-based systems systems from event logs in which the interactions are not observable. Assess segments an event log into traces every time it detects longer time delays between messages. Then, it builds models integrating special actions expressing component calls and proposes two strategies to synchronise these models called *Loosely-coupling* and *Decoupling*. The latter returns more general models than the former by allowing the call of any component any time. We consider both strategies in this evaluation. The approach given in (Mariani and Pastore, 2008) can infer models of communicating systems by segmenting an event log for each component of the system under learning and by applying kbehavior on the trace sets to generate models. Finally, CSight only takes trace sets, one set for each component.

The study has hence been conducted on several use cases with several configurations. Firstly, from a set of 7 devices (3 sensors, 2 gateways, 2 actuators), we built 6 IoT systems denoted $Conf1$ to $Conf6$, by varying the number of devices and the behaviours of the gateway(s) after the receipt of data from the sensors. One or two gateways can be used , themselves interconnected to at least two sensors and one actuator. In each configuration, the components follow a known dependency scheme, which will be used to evaluate RQ3. We also extracted a log from another IoT system denoted $Conf7$, where 8 sensors send data to a gateway. We were unable to build models from any of these configurations with CSight after 5 hours of computation, which was our limit for each experiment. We observed that the first steps of CSight were achieved, but these were always followed by successive time-outs while the model refinement step. Therefore, to compare CSight and CkTail, we took back two trace sets available with the CSight implementation. The first one was extracted from TCP logs, and the second one from logs of the AlternatingBit protocol. As we have trace sets instead of logs with these two use cases, we only compare CSight with the third step of CkTail.

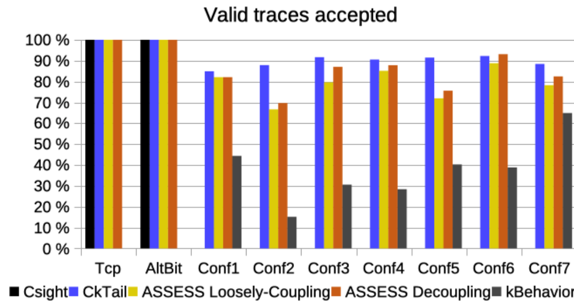

Figure 2: Percentage of valid traces accepted by the models generated with CkTail, Assess, Lfkbehavior.

4.2 RQ1: Can CkTail Derive Models that Capture Correct Behaviours of SUL?

To measure model correctness, we used a cross validation process, which partitions the event logs of every configuration Conf1 to 7 in one training set for the model generation and one testing set. We limited the process to one round, as segmenting an event log cannot arbitrarily be done: we choose to separate an event log into two parts with an approximative ratio of 70% and 30%, taking care not to separate actions that belong to the same session thanks to our knowledge of the configurations. To avoid any bias, model correctness is evaluated by generating valid trace sets from the testing set by considering 3 ways used by the 3 tools. A first trace set is generated by calling Algorithm 1. A second trace set is obtained by splitting the event log every time a long time delay is detected among the messages (as in Assess). The last trace set is achieved by extracting the messages of the event log that share the same component identification (as in LFkbehavior). We obtained around 65 to 285 valid traces for Conf 1 to 7.

Results: Figure 2 shows the percentage of valid traces accepted for each configuration and tool. The bar-diagram firstly shows that the two configurations TCP and AlternatingBit are not sufficient to efficiently compare CkTail and CSight as the models given by both tools accept all the valid traces. But these have few states and the valid trace sets are small. As stated earlier, we were unable to apply CSight on larger trace sets. With the other configurations, the models that accept the most of valid traces are always those generated by CkTail. These models accept an average of 91.90% of valid traces. The models built by Assess tend to have close results as they accept 83.61% of the valid traces with the Loosely-coupling strategy and 86.42% with the Decoupling strategy. The models given by LFkbehavior accept an average of 37.48% of valid traces in our experiments.

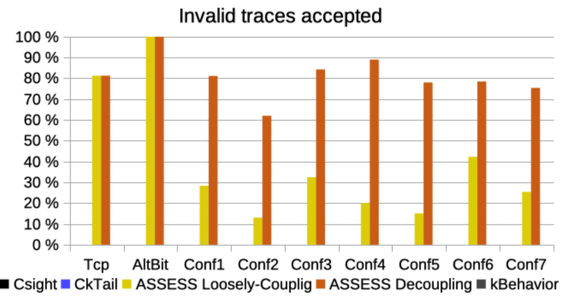

Figure 3: Percentage of invalid traces accepted for each method and configuration.

4.3 RQ2: Can CkTail Build Models that Accept Abnormal Behaviours?

We evaluated how the models generated previously accept invalid traces. The latter were generated by applying these mutations on the valid traces used in RQ1: repetition of actions, inversion of a request with its associated response, and permutation of one request in a sequence of nested requests. We built about 100 invalid traces for the configurations Conf1 to Conf7, and 20 invalid traces for TCP and AlternatingBit. Then, we measured the proportions of invalid traces accepted by the models inferred from the same configurations and training sets used for RQ1.

Results: Figure 3 shows the proportions of invalid traces accepted by the models given by each tool in each configuration. The bar-diagram reveals that only the models produced by Assess accept invalid behaviours. The Decoupling strategy of Assess gives the highest percentages (up to 80%). The main reason behind these results is that Assess is not designed to learn models from message exchanged among components. After studying the models, we indeed observed that many request/response pairs are separated into different models; as the Assess strategies allow repetitive component calls, the models accept successive requests or responses. It results that these models accept many invalid traces.

The results given with RQ1 and RQ2 tend to show that the models produced by CkTail offer the best precision: not only they accept the highest ratio of valid traces, but also reject all the invalid ones (as CSight and LFkbehavior).

4.4 RQ3: Is CkTail Able to Detect Accurate Dependencies among Components?

As CkTail is the only model learning tool able to infer dependency graphs, we chose to evaluate this feature by comparing the DAGs returned by CkTail to the dependency graphs we manually built with our knowledge of the systems under learning. For this study, we took back the configurations for which we have event logs. We listed the component dependency relations found by CkTail and studied them to also provide the incorrect dependencies.

Table 3: Recall and Precision of CkTail to detect component dependencies; Recall is the percentage of the real dependencies that are detected; Precision is the percentage of detected dependencies that are correct.

	Recall of the dependencies found by CkTail	Precision of the dependencies found by CkTail
*Conf*1	100%	100%
*Conf*2	61.5%	100%
*Conf*3	83%	100%
*Conf*4	73%	100%
*Conf*5	64%	100%
*Conf*6	80%	100%
*Conf*7	100%	100%

Results: Table 3 shows the recall and precision of the component dependencies detected by CkTail. On average, CkTail finds 80.27% of the real dependencies and all of them are correct. The missing dependencies (recall below 100%) stem from Definition 3, which strictly specifies what a dependency is in our context, but also restricts the possible dependency patterns to avoid ambiguity. We indeed observed in these experiments that CkTail was able to find several chains of messages sharing the same data addressed to the same component c at the same time, meaning there are several components that potentially have a dependency with c. But, as we are unable to state which component relationship is right, our algorithm leaved them aside.

5 RELATED WORK

We observed in the literature that few approaches were proposed to learn models from communicating systems. Groz et al. introduced in (Groz et al., 2008) an algorithm to generate a controllable approximation of components through active testing. This kind of active technique implies that the system is testable and can be queried. The learning of the components is done in isolation. A recent work lifts this constraint by testing a system with unknown components by means of a SAT solving method (Petrenko and Avellaneda, 2019). In contrast, our approach is passive, and only learns models from logs. Requirements are hence quite different.

Mariani et al. proposed in (Mariani and Pastore, 2008) an automatic detection of failures in log files by means of model learning. Their approach segments an event log with two strategies: per component or per user. The former generates one model for each component. Our evaluation shows that the trace segmentation algorithm of CkTail, which relies on more properties, improves the model precision.

CSight (Beschastnikh et al., 2014) is another tool specialised in the model learning of communicating systems, where components exchange messages through synchronous channels. It is assumed that both the channels and components are known. Besides, CSight requires specific trace sets, which are segmented with one subset by component. CSight follows five stages: 1) log parsing and mining of invariants 2) generation of a concrete FSM that captures the functioning of the whole system by recomposing the traces of the components; 3) generation of a more concise abstract FSM; 4) model refinement with invariants that must hold in models, and 5) generation of Communicating FSM (CFSM). With CkTail, we do not assume that the trace sets are already prepared. We are given an event log from which CkTail tries to detect sessions. Regarding the model generation, CSight allows the use of internal actions (not used in the communications). The current requirements of CkTail prevent from considering internal actions to build models, but we believe that this feature could be added to CkTail in a future work. In theory, CSight should yield more precise models than those given by CkTail because CFMS are refined by checking the satisfiability of invariant. But, in practice, we observed that invariant mining and satisfiability checking are both costly and prevent CSight from taking as input medium to large trace sets.

Compared to the previous approaches, CkTail also has the capability of detecting component dependencies and expresses them with DAGs.

Prior to this paper, we proposed in (Salva and Blot, 2019) the approach and tool called Assess. Its assumptions are different from those required with CkTail or CSight. The main difference lies in the fact that the communications among components are assumed hidden (not available in event logs). Therefore, Assess tries to detect implicit calls of components and adds new synchronisation actions in the models to ex-

press them. Its algorithm is hence specific to this assumption. We compared CkTail with Assess and, as expected, we showed that Assess builds imprecise models when event logs include communications.

6 CONCLUSION

This paper has proposed CkTail, an approach that learns models of communicating systems from event logs. Our algorithm improves the model precision by integrating the identification of dependency relations among components and by better detecting sessions in event logs to extract traces. Unlike CSight, which targets the same kind of systems, CkTail requires as inputs one event log only. Then, it builds execution traces while trying to recognise complete sessions with respect to 4 constraints, whereas the other approaches rely on one or two rules for the trace segmentation. The constraints used by CkTail are specifically related to communicating systems and restrict the trace generation w.r.t. the association of requests/responses, time delay, data dependency, component identification. Besides, CkTail infers DAGs showing the component dependencies. They offer another viewpoint of the component interactions and system architecture, and they may be used to different purposes, e.g., testability measurement, or security analysis.

As future work, we firstly plan to evaluate CkTail on further kinds of systems, e.g., Web service compositions. The trace analysis step relies upon some assumptions for finding sessions in event logs when these are not identified by means of a session mechanism. But, if sessions are clearly identified in messages, these assumptions can be relaxed and the algorithm reduced. We will investigate this possibility in a future work to redesign the first step of CkTail so that it also supports session identification.

ACKNOWLEDGEMENT

Research supported by the French Project VASOC (Auvergne-Rhône-Alpes Region): https://vasoc.limos.fr/

REFERENCES

Ammons, G., Bodík, R., and Larus, J. R. (2002). Mining specifications. *SIGPLAN Not.*, 37(1):4–16.

Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. (2014). Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA. ACM.

Biermann, A. and Feldman, J. (1972). On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597.

Fu, Q., Lou, J.-G., Wang, Y., and Li, J. (2009). Execution anomaly detection in distributed systems through unstructured log analysis. *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158.

Groz, R., Li, K., Petrenko, A., and Shahbaz, M. (2008). Modular system verification by inference, testing and reachability analysis. In Suzuki, K., Higashino, T., Ulrich, A., and Hasegawa, T., editors, *Testing of Software and Communicating Systems*, pages 216–233, Berlin, Heidelberg. Springer Berlin Heidelberg.

Krka, I., Brun, Y., Popescu, D., Garcia, J., and Medvidovic, N. (2010). Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 179–182, New York, NY, USA. ACM.

Lo, D., Mariani, L., and Santoro, M. (2012). Learning extended fsa from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063 – 2076. Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011).

Lorenzoli, D., Mariani, L., and Pezzè, M. (2008). Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 501–510, New York, NY, USA. ACM.

Makanju, A., Zincir-Heywood, A. N., and Milios, E. E. (2012). A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936.

Mariani, L. and Pastore, F. (2008). Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126.

Mariani, L., Pastore, F., and Pezze, M. (2011). Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37(4):486–508.

Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., and Sasnauskas, R. (2018). A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 167–177, New York, NY, USA. ACM.

Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., and Brun, Y. (2014). Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 19–30, New York, NY, USA. ACM.

Pastore, F., Micucci, D., and Mariani, L. (2017). Timed k-tail: Automatic inference of timed automata. In *2017*

IEEE International Conference on Software Testing, Verification and Validation (ICST), pages 401–411.

Petrenko, A. and Avellaneda, F. (2019). Learning communicating state machines. In *Tests and Proofs - 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, pages 112–128.

Salva, S. and Blot, E. (2019). Reverse engineering behavioural models of iot devices. In *31st International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Lisbon, Portugal.

Salva, S. and Durand, W. (2015). Autofunk, a fast and scalable framework for building formal models from production systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 193–204, New York, NY, USA. ACM.

Vaarandi, R. and Pihelgas, M. (2015). Logcluster - a data clustering and pattern mining algorithm for event logs. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 1–7.

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., and Lyu, M. R. (2018). Tools and benchmarks for automated log parsing. *CoRR*, abs/1811.03509.

SCIENCE AND TECHNOLOGY PUBLICATIONS