Noname manuscript No.
(will be inserted by the editor)

Learning of Behavioural Models and Dependency Graphs for Communicating Systems with CkTailv2

Sébastien Salva · Elliott Blot

Received: date / Revised version: date

Abstract This paper presents a model learning approach to recover models from event logs for communicating systems. We refer here to systems made up of components interacting with each other by data networks and whose communications can be monitored, e.g., Internet of Things (IoT) systems, distributed applications or Web service compositions. Our approach, called CkTailv2, is specialised in the generation of behavioural models along with dependency graphs. It generates one Input Output Labelled Transition System (IOLTS) for every component participating in the communications and one graph illustrating the directional dependencies with the other components. These models can help engineers better and quicker understand how a communicating system behaves and is structured. They can also be used for bug detection or for test generation. Compared to other model learning approaches specialised for communicating systems, CkTailv2 improves the precision of the generated models by integrating algorithms that better recognise sessions in event logs. CkTailv2 revisits and extends a first approach by simplifying the set of requirements and assumptions in order to increase its applicability on communicating systems. It now integrates two new trace extraction algorithms: the former segments event logs into traces by trying to detect sessions; the latter assumes event logs to include session identifiers and allows to quicker generate models. We report experimental results obtained from 10 case studies and show that CkTailv2 has the capability of producing precise models in reasonable time delays.

Keywords Model learning; Event logs; Communicating systems; IOLTS; Dependency graphs.

1 Introduction

Generally speaking, behavioural models are abstract representations of systems, which describe how a system responds to external and internal events. It is well-established that such models offer a lot of benefits such as, improving system understanding, preserving knowledge, simulating systems before the code generations, detecting bugs, or automating the test case generation. In the Industry, models are often neglected though. The main concerns are often related to the difficulty of writing models, which is known as a hard and error-prone task, and to the problem of keeping these models up-to-date, especially over the long term.

Model learning is receiving growing attention as a solution to help device models as state machines. These models, which capture system behaviours, can be considered as documentation or exploited in some software engineering stages, e.g. robustness or security testing. Over the last decade, there has been an extensive body of work in this field, making emerge two main categories of approaches called active and passive model learning. Such approaches infer behavioural models of systems seen as a black-boxes, either by interacting with them (active approaches) or by analysing a set of execution traces resulting from monitoring (passive approaches). More recently, some papers proposed innovative model learning algorithms and tools, which have the capabilities to infer symbolic models [28], resource-aware models

Sébastien Salva
LIMOS - UMR CNRS 6158
University Clermont Auvergne, France
E-mail: sebastien.salva@uca.fr

Elliott Blot
LIMOS - UMR CNRS 6158
University Clermont Auvergne, France
E-mail: elliott.blot@uca.fr

[8, 30], timed models [31], and which can be applied to more and more complex systems.

We however observed that a few works [9, 26, 32] focused on the learning of models for communicating systems. Yet, these systems are more and more omnipresent in our daily life, especially with the emergence of Internet of Things (IoT) systems. Model learning would greatly ease the understanding and analysis of communicating systems. For instance, the generation of models expressing the behaviours of every component could help engineers to quicker understand the functioning of the whole system and would assist them in the bug or vulnerability detection. We also noticed that several issues remain open in the previous approaches. For instance, the active technique given in [32] requires to know the system topology in advance and only supports accessible and testable components to build models. But, we have often observed that many communicating systems integrate untestable components. For instance, an autonomous component that continuously delivers messages is uncontrollable and hence cannot be experimented to get observations. The two other papers propose passive approaches, which do not rely on these requirements. Instead, they analyse execution traces to recover behaviours. In order to build precise models, one key point is to be able to recognise sessions in event logs, i.e. 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. Unfortunately, these approaches cannot extract sessions. These observations motivated us to present a first approach and tool called Communicating system k-Tail, shortened CkTailv1 [36]. To design it, we choose to extend the k-Tail learning algorithm [10] with the capability to build one model called Input Output Labelled Transition System (IOLTS) for every component of a communicating system under learning. k-Tail is well-known to quickly build generalised models from traces, but it is unable to take into account the notion of component and to construct models from event logs. Furthermore, CkTailv1 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 showed that CkTailv1 builds more precise models than the two previous passive approaches, but we also concluded that its requirements and assumptions are still too restrictive to be practical.

Contributions: this paper presents an extension of CkTailv1, simply called CkTailv2, and the related tool. This new approach aims at relaxing some requirements of CkTailv1 for targeting more communicating systems. CkTailv2 indeed accepts event logs having communication and non-communication events, the latter being often used to keep track of debug outputs or errors. Event logs can now integrate requests followed by an unlimited amount of responses. Besides, CkTailv2 relies on two session extraction algorithms. The former segments event logs by trying to detect sessions with respect to constraints related to the request-response pattern, the recognition of nested requests, time delays and data dependency among components. The latter assumes event logs to include session identifiers. This assumption simplifies the event log segmentation and allows a quicker model generation.

This paper also provides a detailed empirical evaluation, which investigates the precision of the models derived by CkTailv2 and its performance in terms of execution times. This empirical evaluation was carried out on event logs collected from 10 case studies and compares our implementation of CkTailv2 against three other tools namely CSight, the algorithm given in [26] and CkTailv1. This evaluation shows that CkTailv2 infers more precise models than the three previously approaches, in reasonable time delays.

In summary, the major contributions of this paper are:

- the presentation of the CkTailv2 tool and approach, which generates behavioural models and dependency graphs for every component of a communicating system from event logs,
- the design of two new algorithms allowing to better recognise sessions in event logs, and hence to build more precise models,
- the implementation of the approach publicly available in [12] and an evaluation that compares CkTailv2 with CSight, the approach proposed in [26] and CkTailv1.

Paper organisation: Section 2 discusses related work and presents our motivations. We provide an overview of our tool along with its capability of inferring models of communicating systems with a concrete example of IoT system in Section 3. The CkTailv2's algorithms are detailed in Section 4. We recall some basic definitions about the IOLTS model and we describe the four steps of the approach. Section 5 examines experimental results and discusses about the threats to validity. Section 6 summarises our contributions and draws some perspectives for future work.

2 Related Work

Model learning can be defined as *a set of methods that recover a specification by gathering and analysing system executions and concisely summarising the frequent* *interaction patterns as state machines that capture the system behaviour* [4]. Model learning algorithms can be organised into two main categories: active and passive approaches. Both categories are discussed below.

2.1 Active Model Learning

In this first category, systems are repeatedly queried (often with tests) to collect positive or negative observations, which are analysed and generalised to produce models [3, 5, 7, 15, 21, 22, 33]. Most of the active techniques have been conceived upon two concepts, the $\mathcal{L}^*$ algorithm [5] and incremental learning [15]. This model learning category is actively studied to make the approaches more effective and efficient. Among the possible research directions, some works recently proposed optimisations to reduce the query number [2], while others tackled systems having specific constraints [21].

Some active model learning approaches have been proposed for communicating systems. Groz et al. introduced an algorithm to generate a controllable approximation of components through active testing [19]. 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 [32]. Tappler et al. also proposed a model-based testing technique for IoT systems [41]. This technique is based on the generation of models from multiple implementations of a common specification, which are later pair-wise cross-checked for equivalence. Any counterexample to equivalence is flagged as suspicious and has to be analysed manually.

2.2 Passive Model Learning

The second category includes the techniques that passively recover models from a given set of samples, e.g., a set of execution traces. These are said passive as there is no direct interaction with the system under learning. Models are often generated by encoding sample sets with state diagrams whose equivalent states are merged. For instance, the k-Tail approach [10] merges the states having the same k-future, i.e. the same event sequences having the maximum length k, which all are accepted by the two states. k-Tail has been later enhanced with Gk-tail to generate Extended Finite State Machines encoding data constraints [24]. Other approaches also enhance k-Tail to build more precise models [8, 30, 31]. kBehavior [27] is another kind of approach that generates models from a set of traces by taking every trace one after the other and by completing a finite-state automaton in such a way that it now accepts the trace. These previous passive algorithms usually yield big models, which may quickly become unreadable.

Some passive approaches dedicated to communicating systems have also been proposed. Mariani et al. proposed in [26] an automatic detection of failures in log files by means of model learning. This work extends kBehavior to support events combined with data. It segments an event log with two strategies: per component or per user. The former, which can be used with communicating systems, generates one model for each component. CSight [9] 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 Finite State Machine (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 FSMs, and 5) generation of Communicating FSM.

2.3 Key Observations and Motivations

After having studied the literature, we have firstly observed that few papers and tools tackled the model generation of component based systems or communicating systems. As stated in the introduction, the main concerns of the active model learning techniques are that the component topology must be known in advance, and that all the components must be reachable, testable and resettable many times. As a consequence, active learning can be currently applied on a limited amount of systems. As for passive techniques, the approaches [9, 26] have paved the way, however, there is still room of improvements to relax the approach requirements and to infer precise models. Besides, we have observed that the generation and use of invariants to make models more precise also limits learning to small trace sets only in practice. For instance, the invariant mining and satisfiability checking used in CSight are both costly and prevent the tool from taking as input medium to large trace sets.

We have proposed in [37] a passive model learning algorithm for component-based systems, which builds one model per component to avoid the generation of large and unreadable models. This approach is specialised to IoT systems with an algorithm called Assess [34]. The requirements considered in these approaches are different from those of 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 component calls and adds new synchronisation actions in models. Its algorithm is hence specific to this assumption. Then, we have proposed CkTailv1 [36] to generate models of communicating systems. In short, the novelty proposed by CkTailv1 lies in its capability of detecting sessions in event logs. Indeed, CSight needs sessions put in separate sets but does not provide a way to generate them from event logs. The work proposed in [27] offers the possibility to segment event logs with several strategies. One of them allows to extract the session of every component on condition that the events include component identifiers.

We showed that CkTailv1 builds more precise models than the other approaches by better recognising sessions, but we also concluded that its requirements are too restrictive to be widely used. Indeed, CkTailv1 requires event logs comprising communication events only in such a way that each request has to be followed by one response only. CkTailv2 aims at relaxing some of these assumptions and integrates two new trace extraction algorithms to support more communicating systems.

3 CkTailv2 Tool and Approach Presentation

CkTailv2 is implemented in Java and is released as open source in [12]. The tool takes as inputs an event log collected from a communicating system and a file including regular expressions used to format the event log. It returns two kinds of models. The behaviours of each component of the system under learning are encoded with one IOLTS. Intuitively, an IOLTS expresses here the interactions of one component c with the others along with the non-communication actions of c. Besides, CkTailv2 generates dependency graphs, given under the form of Direct Acyclic Graphs (DAGs). Each component has its own DAG capturing its dependencies towards other components. Such graphs help better comprehend the architecture of the whole system. They complement the IOLTSs by offering another viewpoint of the component interactions and they might be used to different purposes, e.g., testability measurement, or security analysis. Once generated, CkTailv2 stores these models into two folders containing files saved in the DOT format. We chose this format since it is based upon a well-known plain text graph description language that can be translated into graphics formats, e.g., PDF.

We provide below the requirements of CkTailv2, an over-view of its architecture and functioning along with an example of model generation.

3.1 CkTailv2 Requirements

The capability of CkTailv2 of inferring models depends on several realistic assumptions made on a system under learning denoted SUL:

- **A1 Event log:** we consider the components of SUL as black-boxes (no access to firmware, code, data stored on the device, etc.). The communications among the components can be monitored, e.g., on components, on servers, gateways, or by means of wireless sniffers. Event logs are collected in a synchronous environment made up of synchronous communications. Besides, these events are ordered by means of timestamps given by a global clock. At the end of the monitoring process, we consider having one event log;
- **A2 Event content:** components produce communication events or non-communication events. Both kinds of events include parameter assignments allowing to identify the source and the destination of each event. For non-communication events, both the source and the destination refer to the same component that has produced the event. Besides, a communication event can be identified either as a request or a response;
- **A3 Device collaboration:** components can run in parallel and communicate with each other. To learn precise models, we want to recognise sessions of the system in event logs. We consider two exclusive cases:
  - **A31:** the components of SUL follow this strict behaviour: they cannot run multiple instances; requests are processed by a component on a first-come, first served basis. Besides, components follow the request –response exchange pattern (a response is associated to one request, a request is associated to one or more responses), or
  - **A32:** the events that belong to the same session are identified by a parameter assignment.

The session recognition mentioned in A3 helps extract traces expressing complete behaviours of SUL, i.e. disjoint action sequences starting from one of its initial states and ending in one of its final states. A32 represents the classical assumption stating that messages include an identifier allowing to observe whole collaborations among components. Usually, the session identification strongly facilitates the trace extraction. Unfortunately, we have observed that this technique is seldom adopted with communicating systems. When it is not used, we restrict the functioning of SUL with A31 to be able to recognise sessions. We have observed that this assumption can be applied with many wireless or IoT systems.

3.2 CkTailv2 Overview

CkTailv2 is organised into four-steps, illustrated in Figure 1. Initially, the user gives as inputs an event log collected from SUL along with regular expressions. The latter are used to format the event log into a sequence S of actions of the form $a(\alpha)$ with a a label and α some parameter assignments. In accordance with the assumptions A1-A3, the event log formatting allows to highlight some information such as timestamps, or the sources and destinations of the messages (request or response).

Execution traces are extracted from S by means of two algorithms, which rely either on the assumption A31 or A32. In short, if the actions include session identifiers, allowing to directly recognise sessions in S (A32), then we call an algorithm that extracts traces by means of these identifiers. Otherwise, we use a more complex algorithm, which aims at recognising sessions in S with respect to constraints derived from the assumptions A1-A31. Both algorithms return a trace set denoted $Traces$(SUL). In the meantime, these algorithms detect dependencies among the components of SUL and build a dependency set denoted $Deps$(SUL).

Then, the third step of CkTailv2 derives dependency graphs from $Deps$(SUL). In its last step, CkTailv2 generates one IOLTS for every component of SUL with three sub-steps called "4A Trace partitioning", "4B IOLTS Generation" and "4C IOLTS Generalisation". The latter calls the k-Tail approach, which is a model learning technique used to reduce IOLTSs by merging equivalent states.

3.3 Model Learning Example

Before describing the CkTailv2's steps, let us illustrate them with a motivating example of model generation. Figure 2 shows a part of an event log collected from an IoT system made up of devices and of two gateways. The events are formatted by means of regular expressions to produce actions. The regular expression example of Figure 2 extracts from HTTP requests a label equals to the URI along with some parameters. Figure 3 depicts an example of sequence of 15 actions obtained after the first step of CkTailv2. The first four actions are derived from the HTTP messages of Figure 2. As required, these actions indicate the sources and destinations of the messages with the parameters $from$ and to. The other parameter assignments capture acknowledgements or sensor data, e.g., a temperature value with svalue:=68 or a level of luminance with svalue:=1000. We can observe from these actions that the IoT system SUL is made up of 6 components. But interpreting their interactions and what they do is still tricky because of lack of readability.

Traces are now extracted from the action sequence S of Figure 3 by the second step of CkTailv2. It covers and segments S while trying to recognise sessions. In our example, no session identifier is found in the actions. As a consequence, CkTailv2 uses an algorithm that tries to recover sessions with respect to the assumption A31. To be integrated in the algorithm, we formulated this assumption with five constraints expressing what a session is and when keeping an action to a current session. These constraints are detailed in Section 4.3 and summarised as follows: C1: a response is always associated to the last observed request sharing the same communicating components; C2: successive responses are always associated to the related request; C3: nested requests (a request to a component that also performs another request before giving a response) are always kept together in a session; C4: a session gathers messages exchanged between components interacting together in a limited time delay and all the messages capturing a data dependency between two components; C5: a non-communication event is kept is the current session also with respect to time delay and data dependency. Figure 4 gives the trace set $Traces$(SUL) obtained from the action sequence of Figure 3 with this algorithm. For sake of readability, the parameter assignment are concealed in the figure. We observe that it has kept together the related requests and responses, and the nested requests req6 req7. Here, our algorithm has only detected one distinctive longer time interval between the two actions resp5 req6, which implicitly shows that a session ends at resp5 and that a new one begins at req6.

While actions are covered to extract traces, the component interactions are also analysed by CkTailv2 for detecting component dependencies. These dependencies are given under the form of component lists $c_1c_2 \dots c_k$ expressing that a component c_1 depends on a component c_2, which itself depends on another component and so on. The set $Deps$(SUL) gathers these component lists. The component dependency is defined in Section 4.5. Figure 4 shows the set $Deps$(SUL) inferred from our example. Most of the dependencies between pairs of components stem from requests. The component sequence G1G2d3 is detected from the nested requests req6 req7. Four data dependencies are also detected between d2d1, G2d1, d4d1, (with the data svalue:=68) and d3G1 (with the data cmd:=status).

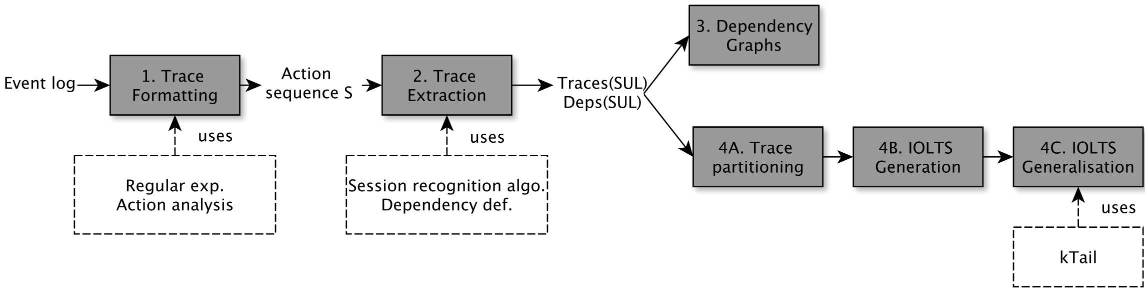

Fig. 1 Model learning of communicating systems with the CkTail approach

```
Jan 20, 2020 09:56:24.225
     CET;Host=d1;Dest=G1;Protocol=HTTP;Verb=GET
     Uri=/req1?svalue=68.00 HTTP/1.1;
Jan 20, 2020 09:56:24.682
     CET;Host=G1;Dest=d1;Protocol=HTTP;HTTP/1.1
     status=200 response=OK;
Jan 20, 2020 09:56:25.153
     CET;Host=G1;Dest=d2;Protocol=HTTP;Verb=GET
     Uri=/req2?svalue=68.00 HTTP/1.1;
Jan 20, 2020 09:56:25.318
     CET;Host=d2;Dest=G1;Protocol=HTTP;HTTP/1.1
     status=200 response=OK data=done;

Example of regular expression:
^(?<date>\w{3} \d{2}, \d{4} \d{2}:\d{2}:\d{2}.\d{3})
\s(CET);(?<param1>(\w+=\w\d));(?<param2>(\w+=\w\d));
(?<param3>[^;]+);(?<param4>[^=]+=[A-Z]{3,4})\s(Uri=)
(?<label>[^?]+)[?](?<param5>(\w+=\d{2}\.\d{2}))\s
HTTP/1.1;$
```

Fig. 2 Example of 4 HTTP messages collected from an IoT system. The regular expression retrieves a label and 5 parameters here. The label expression will be the label of the action in the action sequence S

The CkTail's third step generates dependency graphs. It derives DAGs from the set $Deps$(SUL) and computes their transitive closures. Figure 5 illustrates the dependency graphs obtained in our example.

The fourth step of CkTailv2 lifts traces to the level of IOLTSs. In the step *4A Trace partitioning*, CkTailv2 builds one trace set for every component of SUL. It begins by doubling every communication action to give a pair of output/input actions by separating the notion of source/destination. The non-communication actions are marked as outputs. $Traces$(SUL) is then 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. If we take back our example, Figure 6 gives the new trace sets composed of sequences of input and output actions derived from the set $Trace$(SUL) of Figure 4. As this system is made up of 6 components, $Traces$(SUL) is partitioned into 6 subsets.

```
req1(from:=d1,to:=G1,svalue:=68,time:=
09:56:24.225)
resp1(from:=G1,to:=d1,content:=ok, time:=
09:56:24.682)
req2(from:=G1,to:=d2,svalue:=68, time:=
09:56:25.153)
resp2(from:=d2,to:=G1,content:=done,
time:=09:56:25.318)
req3(from:=G1,to:=G2,svalue:=68, time:=
09:56:26.267)
req1(from:=d1,to:=G1,svalue:=68, time:=
09:56:27.369)
resp3(from:=G2,to:=G1,content:=ok, time:=
09:56:27.371)
resp1(from:=G1,to:=d1,content:=ok, time:=
09:56:27.720)
req5(from:=G2,to:=d4,svalue:=68, time:=
09:56:27.859)
log(from:=d4,to:=d4,content:=heat-off,
time:=09:56:28.909))
resp5(from:=d4,to:=G2,content:=done,
time:=09:56:28.982)
req6(from:=G1,to:=G2,udevice:=12, cmd:=
status,time:=09:56:35.962)
req7(from:=G2,to:=d3,cmd:=status,GPIO:=1
time:=09:56:35.974)
resp7(from:=d3,to:=G2,svalue:=1000,
time:=09:56:36.846)
resp6(from:=G2,to:=G1,svalue:=1000, time:=
09:56:36.958)
```

Fig. 3 Action sequence of an IoT system. These actions have the form <label><parameter assignments>, the latter expressing the components involved in the communications and data

```
Traces(SUL)={req1resp1req2resp2req3req1resp3
resp1req5logresp5, req6req7resp7resp6}

Deps={ d1G1, G1d2, d2d1, G1G2, G2d4, G2d1,
    d4d1, G1G2d3, G2d3, d3G1 }
```

Fig. 4 Step 2: Traces of SUL and dependency set $Deps$(SUL). The parameter assignments are concealed for readability

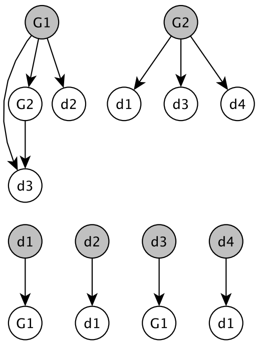

Fig. 5 Step 3: Dependency Graph Generation. Each component has its own dependency graph expressing its directional dependencies with the other components of SUL

```
$T_{d1}$={ !req1 ?resp1 !req1 ?resp1 }
$T_{d2}$={ ?req2 !resp2 }
$T_{d3}$={ ?req7 !resp7 }
$T_{d4}$={ ?req5 !log !resp5 }
$T_{G1}$={ ?req1 !resp1 !req2 ?resp2 !req3
    ?req1 ?resp3 !resp1, !req6 ?resp6 }
$T_{G2}$={ ?req3 !resp3 !req5 ?resp5, ?req6
    !req7 ?resp7 !resp6 }
```

Fig. 6 Step 4A: Trace Partitioning. *Traces*(SUL) is prepared before the IOLTS generation. *Traces*(SUL) is segmented to produce one trace set for every component of SUL

The step *4B IOLTS Generation* transforms every trace 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 6 trace sets, we obtain 6 IOLTSs Ld1-Ld4, LG1, LG2, illustrated in Figure 7. Finally, CkTailv2 applies the k-Tail algorithm to reduce the IOLTS sizes in the step *4C IOLTS Generalisation*. More precisely, it merges the states sharing the same k-future, i.e. the same action sequences having the maximum length k. In our example, with $k := 2$, only the states in white of the IOLTS Ld1 are merged by k-Tail, which produces the IOLTS Ld11.

With these IOLTSs and DAGs, it becomes easier to interpret the behaviour of SUL. In our example, the IOLTSs bring out that the central devices of SUL are G1 and G2, which are the two gateways. The component $d1$ is an active sensor that provides temperature values. These values are sent to two actuators d2 and d4 through the gateways G1 and G2. d3 is a passive sensor (an illuminance light meter) that is queried by G1 through G2, as d3 is directly connected to G2. d4 seems to control a heating system, which is turned off when the temperature reaches 68°F.

Furthermore, as we now have formal models, different kinds of activities may be automatically or semi-automatically conducted to document SUL, to discover defects or more generally to audit SUL. For instance, the European Telecommunications Standards Institute (ETSI) has proposed a general method dedicated to audit large scale, networked systems by undertaking testing and risk assessments [16]. One of the stages of this method corresponds to establishing the context of SUL, which can be partially performed with our tool from event logs. Besides, quality metrics such as testability degrees can be computed from our models [14, 39]. We provide another tool for computing Observability, Controllability and Dependability in [13]. These metrics can be used to deduce which component is testable, or testable in isolation. Other approaches can take these models or transition systems to audit the security of SUL [1, 20, 38, 41].

After this overview of CkTailv2, we will develop its theoretical background along with its algorithms in the next section.

4 The CkTailv2 Approach

Before going to the CkTailv2 step description, we will briefly recall some basic definitions and notations used in the remainder of the paper.

4.1 Preliminary Definitions

As in many works dealing with the modelling of atomic components, e.g., [11, 17], we express the behaviours of components with the well established Labelled Transition System (LTS) model. A LTS is defined in terms of states and transitions labelled by actions, themselves taken from a general action set $\mathcal{L}$, which expresses what happens. The Input Output LTS is an extension of the LTS allowing to better express behaviours with inputs and outputs.

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;
- $\Sigma \subseteq \mathcal{L}$ is the finite set of actions. $\Sigma_I \subseteq \Sigma$ is the countable set of input actions, $\Sigma_O \subseteq \Sigma$ is the countable set of output actions, with $\Sigma_O \cap \Sigma_I = \emptyset$;
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is a finite set of transitions. A transition (q, a, q') is denoted $q \xrightarrow{a} q'$.

We also use the following notations on action sequences. The concatenation of two action sequences σ_1, $\sigma_2 \in \mathcal{L}^*$ is denoted $\sigma_1.\sigma_2$. ϵ denotes the empty sequence. We denote that σ_1 is a subsequence of another sequence σ_2 with $\sigma_1 \preceq \sigma_2$. $final(\sigma)$ denotes the action $a_k(\alpha_k)$

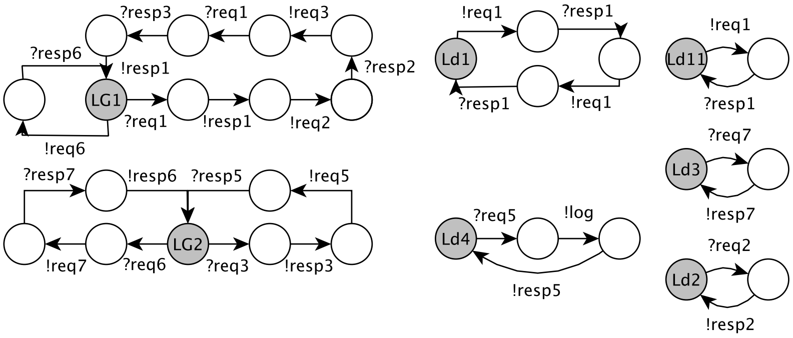

Fig. 7 Steps 4B & C: IOLTS Generation and generalisation with the k-Tail algorithm. Each component has its own IOLTS.

of the sequence $\sigma = a_1(\alpha_1) \dots a_k(\alpha_k)$ or ϵ if $\sigma = \epsilon$. A trace is a finite sequence of observable actions in $\mathcal{L}^*$. The trace of an IOLTS is a sequence $a_0 \dots a_k$ such that $\exists q_{i-1}, q_i, a_i, (1 \leq i \leq k) : q0 \xrightarrow{a_1} q_1 \dots q_{k-1} \xrightarrow{a_k} q_k \in \rightarrow^*$. $Traces(L)$ denote the trace set of the IOLTS 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 an output action composed of the label "switch" followed by a parameter assignment expressing the components involved in the communication and a parameter of the switch command.

We finally will use the following notations on actions to make our algorithms more readable:

- $from(a(\alpha)) = c$ denotes the source of the action;
- $to(a(\alpha)) = c$ denotes the destination;
- $components(a(\alpha)) = \{from(a(\alpha)), to(a(\alpha))\}$;
- $time(a(\alpha)) = t$ returns the timestamp value identifying when $a(\alpha)$ occurred, $time(\epsilon) = +\infty$;
- $isReq(a(\alpha))$, $isResp(a(\alpha))$ are boolean expressions expressing the nature of the message;
- $session(a(\alpha)) = id$ denotes the session identifier when available. Otherwise, $session(a(\alpha)) = \emptyset$.
- $data(a(\alpha)) = \alpha \setminus \{from := c_1; to := c_2, time := t, session := s\}$;

The dependencies among the components of a communicating system are captured with a Directed Acyclic Graph (DAG), where component identifiers are labelled on vertices.

Definition 2 (Directed Acyclic Graph) A DAG Dg is a 2-tuple $\langle V_{Dg}, E_{Dg} \rangle$ where V is the finite set of vertices and E the finite set of edges.
λ denotes a labelling function mapping each vertex $v \in V$ to a label $\lambda(v)$.

4.2 CkTailv2 Step 1: Trace Formatting

Keeping in mind the assumption A1, CkTail takes as input an event log gathering events that are totally ordered by means of their time-stamps. These events are parsed to retrieve the actions performed by SUL and their related data. These actions must have the form $a(\alpha)$ with a a label and α an assignment of parameters and must be compliant with the assumption A2. This formatting is achieved by means of regular expressions given to CkTailv2. Their writing may be performed manually with small to medium event logs, but this activity may quickly become laborious as the log size grows. A way to eliminate or assist users in this intervention is to consider the approaches and tools that automatically mine patterns from log files [18, 25, 26, 29, 42, 43]. These patterns may be used to quickly derive regular expressions.

As events are usually too detailed or specific to their related executions, regular expressions are also a good mean to lift the abstraction level by filtering out some useless actions, or some concrete values in actions.

At the end of this step, we hence assume having a sequence $S \in \mathcal{L}^*$ of actions on the form $a_1(\alpha_1) \dots a_k(\alpha_k)$. The next step of CkTailv2 covers the action sequence S to extract the sub-sequences that capture some sessions of SUL. This step relies either on the assumption A31 or A32 and is hence implemented with two different algorithms presented in the two next sections.

4.3 CkTailv2 Step 2: Trace Extraction Without Session Identifier

The first trace extraction algorithm is founded on the assumptions A1, A2, A31 to interpret communications and to recover sessions in event logs. In particular, with A31, we suppose that sessions are not identified in event logs.

C1	A response $a_i(\alpha_i)$ is always associated to the last request previously observed in σ such that the replier returns the response to the requester which has sent the request.
C2	All the responses associated to the same request are kept in σ.
C3	A request $a_i(\alpha_i)$ that belongs to a chain of nested requests must be kept in the session σ. Two requests req1 and req2 are nested iff the action sequence S includes this form of sequence: req1(from:=c1, to:=c2) req2(from:=c2, to:=c3) resp2(from:=c3, to:=c2) resp1(from:=c2, to:=c1).
C4	A component, which already participated to the session σ, can send a new request $a_i(\alpha_i)$ to another component. This request is kept in σ if C4.1: the session is not timed out, or if C4.2: this request shares data with some previous actions of σ
C5	A non-communication action $a_i(\alpha_i)$ is kept in σ if C5.1: the session is not timed out, or if C5.2: $a_i(\alpha_i)$ shares data with some previous actions of σ

Table 1 Constraints derived from the assumptions A1, A2, A31. When one of these constraints hold, the current action $a_i(\alpha_i)$ is kept in a session σ.

To devise this algorithm, we derived a list of constraints from these assumptions giving the conditions for a sub-sequ-ence of S to be a session. As our algorithms cover the actions of S one after the other, we have formulated these constraints to express whether an action $a_i(\alpha_i)$ of the action sequence $S = a_1(\alpha_1)\dots a_k(\alpha_k) \in \mathcal{L}^*$ belongs to a session denoted σ. Table 1 gathers the five constraints used in our algorithms. C1 and C2 focus on responses, while C3 and C4 deal with requests. C4 is a special constraint expressing when a new request $a_i(\alpha_i)$, sent from a component that has previously participated in the current session, belongs to σ. The choice of keeping this new request in the session depends on two other factors, i.e., time delay and data dependency, with the constraints C4.1 and C4.2. C5 addresses non-communication actions and restricts the session participation as in C4.

To use these constraints in our algorithms, we formulated them with boolean expressions written with the notations given in Section 4.1 completed by these ones:

- KC stands for the set of known components involved in the session σ so far;
- *response*$(a_1(\alpha_1), a(\alpha))$ is the boolean expression *isResp*$(\ a_1(\alpha_1))\wedge$ $from(a_1(\alpha_1)) =$ *to*$(a(\alpha))\wedge$ *to*$(a_1(\ \alpha_1)) = from(a(\alpha))$;
- $Lreq(\sigma)$ denotes the set of sequences of pending requests i.e. the sequences of requests $a_1(\alpha_1)\dots a_k(\alpha_k) \preceq \sigma$ for which responses have not yet been received. $Lreq(\sigma) =_{def} \{a_1(\alpha_1)\dots a_k(\alpha_k) \preceq \sigma \mid isReq(a_i(\ \alpha_i))_{1\leq i\leq k}, \forall a(\alpha) \in \mathcal{L}^* : response(a(\alpha), a_i(\alpha_i)) \implies a_i(\alpha_i)a(\alpha)a_{i+1}(\alpha_{i+1}) \not\preceq \sigma\}$;
- $OLreq(\sigma)$ denotes the set of requests for which a least one response has been received;
- *ontime*$(a(\alpha), \sigma)$ is a boolean expression that returns true if the action $a(\alpha)$ may belong to the session σ with regard to the session duration or session timeout;
- *data-dependency*$(a(\alpha), S, \sigma)$ is a boolean expression that returns true if the request $a(\alpha)$ shares some data with other requests of the session $\sigma \preceq S$. The data dependency is defined in Section 4.5;
- *pendingRequest*(c) is the boolean expression $(\exists\sigma_1 \in Lreq(\sigma), a(\alpha) \in \sigma_1 : c \in components(a(\alpha)))$ that evaluates whether the component c has sent (resp. received) a request and has not yet received (resp. sent) the response.

C1 C2	$\exists!\sigma_r \in Lreq(\sigma) : response(a_i(\alpha_i), final(\sigma_r))\}$ $\exists!\sigma_r \in OLreq(\sigma) : response(a_i(\alpha_i), final(\sigma_r))\}$
C3	$isReq(a_i(\alpha_i)) \wedge Lreq' = \{\sigma_1 \in Lreq(\sigma) \mid from(a_i(\alpha_i)) = to(final(\sigma_1))\} \neq \emptyset\wedge \neg pendingRequest(from(a_i(\alpha_i)))$
C4	$isReq(a_i(\alpha_i))\wedge$ $from(a_i(\alpha_i)) \in \mathrm{KC} \wedge(\forall\sigma_1 \in Lreq(\sigma) : from(a_i(\alpha_i)) \neq to(final(\sigma_1)))\wedge$ $(ontime(a_i(\alpha_i), \sigma) \vee dataDependency(a_i(\alpha_i), S, \sigma))\wedge$ $\neg pendingRequest(from(a_i(\alpha_i)))$
C5	$\neg isReq(a_i(\alpha_i))\wedge$ $\neg isResp(a_i(\alpha_i))\wedge$ $from(a_i(\alpha_i)) \in \mathrm{KC} \wedge$ $(ontime(a_i(\alpha_i), \sigma)\vee$ $dataDependency(a_i(\alpha_i), S, \sigma))$

Table 2 Formalisation of the constraints C1-C5 used in the trace extraction algorithm

From these notations, we formulated the above constraints, listed their boolean terms and studied their possible permutations. We finally kept the constraints expressing that an action $a_i(\alpha_i)$ belongs to the current session when they hold. These are listed in Table 2.

Algorithms 1 and 2 implement the trace extraction. Algorithm 1 calls the procedure *Keep-or-Split* with an action sequence initialised to S. It returns $Traces$(SUL), the final component set C along with the set of component dependencies *Deps*(SUL).

The procedure *Keep-or-Split* covers an action sequence $a_1(\ \alpha_1)\dots a_k(\alpha_k)$ to extract a session σ. The set of known components KC is initialised with the components of the first action $a_1(\alpha_1)$. Then, every action $a_i(\alpha_i)$ is covered to decide whether it is kept in σ (line 8) or not. Given an action $a_i(\alpha_i)$, the procedure *updateOLreq* (Algorithm 2 lines (1-5)) is called to update the set of pending requests $OLreq$ w.r.t. the assumption A31. More precisely, if $a_i(\alpha_i)$ is a new request coming from a component c, then all the previous requests that involve c are removed from $OLreq$ to meet A31 (first come, first served). In the same way, if $a_i(\alpha_i)$ is a response, only the request associated to this response is kept.

Then, the procedure *Keep-or-Split* processes the action $a_i(\alpha_i)$ with the constraints C1-C5. When one of them holds, the action $a_i(\alpha_i)$ is added to the session σ. Besides, the set of known components KC is updated to include the components involved in $a_i(\alpha_i)$. For any other case, the action $a_i(\alpha_i)$ is put into a new action sequence σ_2 (line 27). Once all the actions have been

Algorithm 1: Trace Extraction with A31

```
   input  : Action sequence $S$
   output : $Traces(\mathrm{SUL})$, Component set $C$, Component
            dependency set $Deps(\mathrm{SUL})$
1  $C := Deps(\mathrm{SUL}) := \emptyset$;
2  Keep-or-Split($S$);
3  Procedure $Keep\text{-}or\text{-}Split(a_1(\alpha_1)\dots a_k(\alpha_k))$ is
4      $\sigma := \sigma_2 := \epsilon$;
5      $Lreq(\sigma) := OLreq(\sigma) := \emptyset$;
6      $\mathrm{KC} := components(a_1(\alpha_1))$;
7      $i := 1$;
8      while $i \leq k$ do
9          $updateOLreq(a_i(\alpha_i))$;
10         case $C1$ $true$ do
11             $\sigma := \sigma.a_i(\alpha_i)$; $Trim(\sigma_r)$;
12             $\mathrm{KC} := \mathrm{KC} \cup components(a_i(\alpha_i))$;
13         case $C1$ $false$ $and$ $C2$ $true$ do
14             $\sigma := \sigma.a_i(\alpha_i)$;
15             $\mathrm{KC} := \mathrm{KC} \cup components(a_i(\alpha_i))$;
16         case $C3$ $true$ do
17             $\sigma := \sigma.a_i(\alpha_i)$;
18             $Extend(\sigma_r, a_i(\alpha_i))$;
19             $\mathrm{KC} := \mathrm{KC} \cup components(a_i(\alpha_i))$;
20         case $C3$ $false$ $and$ $C4$ $true$ do
21             $\sigma := \sigma.a_i(\alpha_i)$;
22             $Extend(\epsilon, a_i(\alpha_i))$;
23             $\mathrm{KC} := \mathrm{KC} \cup components(a_i(\alpha_i))$;
24         case $C5$ $true$ do
25             $\sigma := \sigma.a_i(\alpha_i)$;
26             $\mathrm{KC} := \mathrm{KC} \cup components(a_i(\alpha_i))$;
27         otherwise do $\sigma_2 := \sigma_2.a_i(\alpha_i)$ ;
28         i++;
29     $Traces(\mathrm{SUL}) := Traces(\mathrm{SUL}) \cup \{\sigma\}$;
30     $C := C \cup \mathrm{KC}$;
31     if $\sigma_2 \neq \epsilon$ then
32         Keep-or-Split($\sigma_2$);
33 END;
```

Algorithm 2:

```
1  Procedure $updateOLreq(a_i(\alpha_i))$ is
2      if $isReq(a_i(\alpha_i))$ then
3          $OLreq(\sigma) := OLreq(\sigma) \setminus \{a(\alpha) \in OLreq \mid$
             $from(a_i(\alpha_i)) \in components(a(\alpha))\}$;
4      else if $isResp(a_i(\alpha_i))$ then
5          $Lr := \{a(\alpha) \in OLreq(\sigma) \mid from(a_i(\alpha_i) =$
             $to(a(\alpha))\}$
             $OLreq(\sigma) := OLreq(\sigma) \setminus \{a(\alpha) \in OLreq(\sigma) \mid$
             $from(a_i(\alpha_i) \in components(a(\alpha))\} \cup Lr$;
6  Procedure $Trim(\sigma_r)$ is
7      $\sigma' := remove(final(\sigma_r))$;
8      $Lreq(\sigma) := Lreq(\sigma) \setminus \{\sigma_r\} \cup \{\sigma'\}$;
9      $OLreq(\sigma) := OLreq(\sigma) \setminus \{a(\alpha) \in OLreq(\sigma) \mid$
         $from(final(\sigma_r)) \in components(a(\alpha))\}$;
10     $OLreq(\sigma) := OLreq(\sigma) \cup \{final(\sigma_r)\}$;
11 Procedure $Extend(\sigma_r, a(\alpha))$ is
12     $\sigma' := \sigma_r.a(\alpha) = a_1(\alpha_1)\dots a_k(\alpha_k)$;
13     $Lreq(\sigma) := Lreq(\sigma) \setminus \{\sigma_r\} \cup \{\sigma'\}$;
14     //Component dependencies
15     $lc := c_1 \dots c_k c_{k+1}$ such that $c_i = from(a_i(\alpha_i))_{(1 \leq i \leq k)}$,
         $c_{k+1} = to(a_k(\alpha_k))$;
16     $Deps(\mathrm{SUL}) := Deps(\mathrm{SUL}) \cup \{lc\}$;
17 Procedure $ontime(a_i(\alpha_i), \sigma)$ is
18     return $(time(a_i(\alpha_i)) - time(final(\sigma)) < T)$;
19 Procedure $data\text{-}dependency(a_i(\alpha_i), S, \sigma)$ is
20     if $\exists \sigma_1 = a_1(\alpha_1)a_2(\alpha_2)\dots a_i(\alpha_i) \preceq S : to(a_i(\alpha_i)) \xrightarrow[\sigma_1]{data}$
         $from(a_1(\alpha_1))$ then
21         $Deps(\mathrm{SUL}) :=$
             $Deps(\mathrm{SUL}) \cup \{to(a_i(\alpha_i)).from(a_1(\alpha_1))\}$;
22         if $\sigma_1 \preceq \sigma.a_i(\alpha_i))$ then
23             return true;
24     return false;
```

covered, σ is added to $Traces(\mathrm{SUL})$ and C is updated with the set of components KC built with this session. If σ_2 is not empty, the procedure *Keep-or-Split(σ_2)* is recursively called to recover other sessions in σ_2 (line 31).

The main difference among the cases C1 to C5 lies in the update of the set of pending requests $Lreq(\sigma)$, with the procedures $Trim$ and $Extend$. The former is called with C1: receipt of a response associated to a list of pending requests σ_r in $Lreq(\sigma)$. $Trim$ is called to remove the last request of σ_r, $final(\sigma_r)$, because a response has been received to this request. $final(\sigma_r)$ is shifted to $OLreq(\sigma)$. The procedure $Extend$ is called with C3 and C4. C3 corresponds to the receipt of a request that belongs to a chain of nested requests $\sigma_r \in Lreq(\sigma)$. $Extend$ is here called to update $Lreq(\sigma)$ with the nested request list $\sigma_r.a_i(\alpha_i)$. C4 stands for the receipt of a new request from a known component. $Extend$ is now called to add the new request $a_i(\alpha_i)$ in $Lreq(\sigma)$. Furthermore, $Extend$ builds the set $Deps(\mathrm{SUL})$ of component lists. This part is detailed in Section 4.5.

The boolean expression $ontime(a(\alpha), \sigma)$, used in C4 and C5, is implemented with the procedure *ontime*. As stated previously *ontime* allows to limit the session duration. Several implementations are possible. We provide an example in Algorithm 2, line (17). This procedure checks whether the time delay between the last received action $a_i(\alpha_i)$ and the previous one in the session σ is lower than a time duration T.

The boolean expression *data-dependency* $(a_i(\alpha_i), S, \sigma)$, also used in C4 and C5, is implemented by the procedure given in Algorithm 2. It checks whether a data dependency exists between the request $a_i(\alpha_i)$ and some requests of the session σ. The notion of dependency among components and this procedure shall be discussed in Section 4.5.

The action sequence of Figure 3 has been converted into $Traces(\mathrm{SUL})$ by means of this algorithm, as no session identifier is available within actions. Here, the trace extraction algorithm has detected that C4 does not hold with the request req6. It has indeed detected, by means of the timestamps, a distinctive longer time interval between the actions resp5 req6, which implicitly suggests that the session timed out. The algorithm has detected two nested requests req6 req7. Besides, several data dependencies have been identified between the requests req1,req2 req3,req5. These requests along with their responses are hence kept together in the same session.

4.4 CkTailv2 Step 2: Trace Extraction With Session Identifier

Algorithm 3: Trace Extraction with A32

```
   input  : Action sequence $S$
   output : $Traces(\mathrm{SUL})$, Component set $C$, Component
            dependency set $Deps(\mathrm{SUL})$
 1 $C := Deps(\mathrm{SUL}) := \emptyset$;
 2 $ID := \{session(a(\alpha)) \mid a(\alpha) \in S\}$;
 3 $Traces(\mathrm{SUL}) := \bigcup_{id \in ID}\{\sigma_{id}\}$ with
     $\sigma_{id} = S \setminus \{a(\alpha) \mid session(a(\alpha)) \neq id\}$;
 4 foreach $\sigma = a_1(\alpha_1) \dots a_k(\alpha_k) \in Traces(\mathrm{SUL})$ do
 5     $S := \sigma$;
 6     Keep-or-Split2($S$);
 7 END;
 8 Procedure Keep-or-Split2($a_1(\alpha_1) \dots a_k(\alpha_k)$) is
 9     $Lreq(\sigma) := OLreq(\sigma) := \emptyset$;
10     $\mathrm{KC} := components(a_1(\alpha_1))$;
11     $i := 1$;
12     while $i \leq k$ do
13         $C := C \cup components(a_i(\alpha_i))$;
14         case C1 true do
15             $Trim(\sigma_r)$;
16         case C3 true do
17             $Extend(\sigma_r, a_i(\alpha_i))$;
18         case C3 false and C4 true do
19             $Extend(\epsilon, a_i(\alpha_i))$;
20         i++;
```

The previous trace extraction algorithm relies on the assumption A31 to extract traces. This second trace extraction algorithm now relies on A32. This assumption, which means that all the events have to include session identifiers, strongly simplifies the recovering of sessions, and hence the trace extraction. But, as we also wish to detect component dependencies, the trace extraction algorithm, given in Algorithm 3, still covers every action of an action sequence S.

Algorithm 3 begins to build $Traces(\mathrm{SUL})$ by extracting from S the sub-sequences of actions having the same session identifier (lines 2-3). Afterwards, it calls the procedure *Keep-or-Split2* for every trace of $Traces(\mathrm{SUL})$ to detect component dependencies (lines 4-6). To this end, this procedure updates the set of pending requests $Lreq(\sigma)$ as previously for every trace σ with the constraints C1, C3, C4 (only these constraints are needed to build $Lreq(\sigma)$). $Lreq(\sigma)$ is updated by means of the procedures $Trim$ and $Extend$, which disclose component dependencies and build the set $Deps(\mathrm{SUL})$.

4.5 CkTailv2 Step 3: Dependency Graph Generation

The notion of component dependency is formulated by means of the three expressions given below. We write c_1 *depends on* c_2, when at least one of these expressions holds.

Definition 3 (Component dependency) Let $c_1, c_2 \in C$, $c_1 \neq c_2$, and $S \in \mathcal{L}^*$. We denote c_1 depends on c_2 iff $(c_1 \xrightarrow[\sigma]{r} c_2) \vee (c_1 \xrightarrow[\sigma]{nr} c_2) \vee (c_1 \xrightarrow[\sigma]{data} c_2)$ with:

1. $c_1 \xrightarrow[\sigma]{r} c_2$ iff $\exists \sigma \preceq S, a(\alpha) \preceq \sigma : isReq(a(\alpha))$, $from(a(\alpha)) = c_1, to(a(\alpha)) = c_2$;
2. $c_1 \xrightarrow[\sigma]{nr} c_2$ iff $\exists \sigma \preceq S$, $a_1(\alpha_1) \dots a_k(\alpha_k) \preceq \sigma :$ $from(a_1(\alpha_1)) = c_1, to(a_k(\alpha_k)) = c_2$, $a_1(\alpha_1) \dots a_k(\alpha_k) \in Lreq(\sigma)$;
3. $c_1 \xrightarrow[\sigma]{data} c_2$ iff $\exists \sigma \preceq S, \alpha \in P : DS(\sigma, c_1, c_2, \alpha)$ and $\forall \sigma' = a_1'(\alpha_1')a_2'(\alpha_2') \dots a_k(\alpha_k) \preceq S : DS(\sigma', c_1, c_2, \alpha) \implies \sigma' \preceq \sigma$, with $DS(a_1(\alpha_1) \dots a_k(\alpha_k) c_1, c_2, \alpha)$ the boolean expression $from(a_1(\alpha_1)) = c_2 \wedge\ to(a_k(\alpha_k)) = c_1 \wedge\ isReq(a_k(\ \alpha_k)) \wedge\ to(a_i(\alpha_i)) = from(a_{i+1}(\ \alpha_{i+1})\ )_{1 \leq i < k)} \wedge \bigcap_{(1 \leq i \leq k)} \alpha_i = \alpha$.

The two first expressions illustrate that a component c_1 depends on another component c_2 when c_1 queries c_2 with a request or by means of successive nested requests of the form $req1(from := c_1, to := c)req2(from := c, to := c_2)$. The last expression deals with data dependency. We say that c_1 depends on c_2 if there is a chain of actions from c_2 ended by a request to c_1 sharing the same data α. More precisely, the third expression holds if a component c_2 has sent an action $a_1(\alpha_1)$ with some data α, if there is a unique sequence $a_1(\alpha_1) \dots a_k(\alpha_k)$ sharing this data and if $a_k(\alpha_k)$ is a request whose destination is c_1. An immediate consequence of this expression is that we do not consider component dependencies when there are several chains of actions all sharing the same data and addressed to the several components. Yet, we can observe that there is a data dependency among components, but we are unable to establish the dependency relations as several options among the components are possible. Because of this ambiguity that may bring false relationships, we prefer to not consider this case.

The component dependencies are detected by the second step of CkTailv2 and are given under the form of component lists $c_1 \dots c_k$ in $Deps(\mathrm{SUL})$. Component dependencies are detected while Algorithms 1 or 3 build traces by means of the procedures $Extend$ and *data-dependency*. The procedure $Extend$ detects the two first component dependency cases of Definition 3. It uses the set of pending requests $Lreq(\sigma)$ to complete the set $Deps(\mathrm{SUL})$. Indeed, the procedure $Extend$ constructs a sequence of $Lreq(\sigma)$ in such a way that it is either one request (Case C4) or a list of nested requests (Case C3). The procedure covers the component sequences $lc = c_1 \dots c_k c_{k+1}$ of $Lreq(\sigma)$ and adds the dependency lists in $Deps(\mathrm{SUL})$ (Algorithm 2, line 15). The procedure *data-dependency*$(a_i(\alpha_i), S, \sigma)$ checks whether the last expression of Definition 3 holds. If there is a unique sequence $a_1(\alpha_1) \dots (a_i(\alpha_i)$ 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 $Deps(\text{SUL})$ (line 21). If this sequence is a subsequence of the current session $\sigma.a_i(\alpha_i)$, then the procedure also returns true to Algorithms 1 and 3 to indicate that this request must be kept in the current session.

It is worth noting that Algorithms 1 and 3 slightly differ in the data dependency detection. Given two components c_1 and c_2, Algorithm 1 checks whether $c_1 \xrightarrow[\sigma]{data} c_2$ holds on the initial action sequence S. It checks that there is a unique chain of actions from c_2 to c_1 in S as it does not know the sessions in advance. Algorithm 3 does the same verification but on every trace σ of $Traces(\text{SUL})$, which represent sessions. As a trace σ is usually much shorter than the action sequence S, $c_1 \xrightarrow[\sigma]{data} c_2$ may be satisfied more frequently. In other terms, Algorithm 3 may detect more component dependencies because the sessions are already given and known.

Figure 4 shows the set $Deps(\text{SUL})$ derived from the action sequence of Figure 3. Most of the component dependencies stem from requests. For instance, the component sequence G1G2d3 is detected from the nested requests req6 req7. Four data dependencies are detected between d2d1, G2d1 d4d1, (with the data svalue:=68) and d3G1 (with cmd :=status).

CkTailv2 implements the generation of dependency graphs from $Deps(\text{SUL})$ with Algorithm 4. The latter partitions $Deps(\ \text{SUL})$ to group the dependency lists starting by the same component into the same subset. This partitioning is performed with the equivalence relation $\sim_c$ on C^* given by $\forall l_1, l_2 \in Deps(\text{SUL})$, with $l_1 = c_1 \dots c_k$, $l_2 = c'_1 \dots c'_k$, $l_1 \sim_c l_2$ iff $c_1 = c'_1$. Given a partition C_i and a component list $l \in C_i$, Algorithm 4 builds a path of the DAG Dg_i such that the nth state is labelled by the nth component of l. Algorithm 4 finally computes the transitive closure of the DAGs to make all component dependencies visible.

The dependency graphs, which are generated from the set $Deps(\text{SUL})$ of Figure 4, are depicted in Figure 5. They reflect another window on the architecture of SUL. Indeed, these graphs show in a readable manner how the components interact together. They also help identify central components that might have a strong negative impact on SUL when they integrate faults.

Algorithm 4: Device Dependency Graphs Generation

```
input  : $Deps(\text{SUL})$
output : Dependency graph set $DG$
foreach $C_i \in Deps(\text{SUL}) / \sim_c$ do
    foreach $c_1 c_2 \dots c_k \in C_i$ do
        add the path $s_{c_1} \rightarrow s_{c_2} \dots s_{c_{k-1}} \rightarrow s_{c_k}$ to $Dg_i$;
    $Dg'_i$ is the transitive closure of $Dg_i$;
    $DG := DG \cup \{Dg'_i\}$;
```

Algorithm 5: IOLTS Generation

```
   input  : $Traces(\text{SUL})$
   output : IOLTSs $L_{c_1} \dots L_{c_k}$
1  $T := \{\}$;
2  foreach $\sigma = a_1(\alpha_1) \dots a_k(\alpha_k) \in Traces(\text{SUL}))$ do
3      $\sigma' := \epsilon$;
4      foreach $a_i(\alpha_i) \preceq \sigma$ do
5          $\sigma' := \sigma'.!a_i(\alpha_i \cup \{id := from(a_i(\alpha_i))\} \setminus \{time := t, session := s\})$;
6          if $isReq(a_i(\alpha_i)) \vee isResp(a_i(\alpha_i)))$ then
7              $\sigma' := \sigma'.?a_i(\alpha_i \cup \{id := to(a_i(\alpha_i))\} \setminus \{time := t, session := s\})$;
8      foreach  $c \in C$ do
9          $T_c := T_c \cup \{\sigma' \setminus \{a(\alpha) \in \sigma' \mid (id := c) \notin \alpha\}$
10 foreach $T_c$ $with$ $c \in C$ do
11     Generate the IOLTS $L_c$ from $T_c$;
12     Merge the equivalent states of $L_c$ with $kTail(k = 2, L_c)$;
```

4.6 CkTailv2 Step 4: IOLTS Generation

This last step, implemented by Algorithm 5, generates one IOLTS for every component in C. The algorithm starts by transforming the traces to integrate the notions of input and output. Given a trace $a_1(\alpha_1) \dots a_k(\alpha_k)$, every action is doubled by separating the component source and destination. The source and the destination are identified by a new assignment on the parameter id added to each action. Besides, the timestamps and session identifiers are removed from the assignments to improve the model generalisation. For a communication action $a_i(\alpha_i)$, this step produces a new trace σ' 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 (lines 5-7). Non-communication actions are marked as outputs. Then, this new trace σ' is segmented into sub-sequences, each capturing the behaviours of one component only (lines 8, 9). The trace set T_c gathers the traces of the component c.

Every trace set T_c is now lifted to the level of IOLTS. A trace $t = a_1(\alpha_1) \dots a_k(\alpha_k) \in T_c$ is transformed into the path $q0 \xrightarrow{a_1(\alpha_1)} q_1 \dots q_{k-1} \xrightarrow{(a_k(\alpha_k)} q0$ 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 5 applies the kTail algorithm to generalise and reduce the IOLTSs by merging the equivalent states having the same k-future. We use $k = 2$ as recommended in [23, 24].

5 Empirical Evaluation

The experiments presented in this section aim to evaluate the capabilities of our algorithms to build models in terms of precision and performance, compared to the approaches allowing to learn models of communicating systems. Prior to this work, we evaluated CkTailv1 along with the tools CSight [9], Assess [35], and the tool suite proposed in [26] based upon the tool kbehavior, which we denote LFkbehavior. Our experimental results, given in [36], showed that Assess requires assumptions that are strongly different than those required by the other tools. The main difference for Assess lies in the fact that the communications among components are assumed hidden (not available in event logs). Assess tries to detect implicit calls of components instead, and completes models with synchronisation actions to express them. When this assumption does not hold, i.e. when we feed Assess with event logs including communication messages, we showed that it builds high imprecise models. Consequently, for this new evaluation, we have chosen to conduct several experimentations on CSight, LFkbehavior, CkTailv1 and CkTailv2 (source code and explanations available in [12]). As our approach uses two distinct trace extraction algorithms, we have chosen to differentiate them with the notations CkTailv2-w/oS (Algorithm 1 without session identifier) and CkTailv2-w/S (Algorithm 3 with session identifiers).

This evaluation aims at investigating the capabilities of our algorithms through the following four questions:

- RQ1: can CkTailv2 infer models that capture correct behaviours of SUL? This question studies the capability of CkTailv2 to build models that accept valid traces of the system compared to CSight, CkTailv1 and LFkbehavior. The valid traces correspond to traces extracted from event logs but not used for the model generation;
- RQ2: do the models inferred by CkTailv2 reject abnormal behaviours? RQ2 studies the capability of CkTailv2 to generate models that reject invalid traces, compared to CSight, CkTailv1 and LFkbehavior. Invalid traces express abnormal behaviours of the system;
- RQ3: is CkTailv2 able to detect accurate dependencies among components? RQ3 investigates the recall and precision of CkTailv2 to detect component dependencies. Recall is here the percentage of the real dependencies that are detected, and precision is the percentage of detected dependencies that are correct;
- RQ4: what is the performance of CkTailv2 to infer models compared to the other tools? How does CkTailv2 scale with the size of the event log?

5.1 Empirical Setup

To generate models, the considered tools impose different assumptions, which we examined before our experiments to avoid any bias. We ran LFkbehavior with the strategy that segments event logs w.r.t. component identifier, as this is the only one that can be applied with communicating systems to build one model per component. CSight doesn't take event log as input but trace sets such that every component is associated to its own trace set. CkTailv1 is more restrictive on the event log content than LFkbehavior and CkTailv2. For CkTailv1, an event log must be exclusively composed of communication events and a request must be associated to one response only.

As a consequence, we have taken into consideration all these differences through experiments conducted on several configurations. We firstly assembled and configured 6 communicating systems from a set of 7 commercial devices (3 sensors, 2 gateways, 2 actuators). Each of these systems contains at least one gateway using the home automation system Domoticz [1], connected to at least two sensors and one actuator. The behaviours of the gateway(s) after the receipt of data from the sensors differ in each configuration. We monitored these systems and collected event logs of about 2200 events. We denote them $Conf1$ to $Conf6$. We also considered 2 other systems made up of other components to avoid giving conclusions on similar systems. The first one has 8 sensors (4 are commercial devices and the others are based upon the open source framework EspEasy [2]) that periodically send data to a Cloud server. The second one corresponds to an IP security camera, which is interconnected to NTP, SMTP and FTP servers. The corresponding event logs are denoted $Conf7$ and $Conf8$ and respectively include 2206 and 1310 events. All these event logs do not include session identifiers. Hence, we manually modified them to compare our algorithms CkTailv2-w/oS and CkTailv2-w/S. The modifications consisted in adding a session identifier in every action with regard to the

[1] https://www.domoticz.com/

[2] https://www.letscontrolit.com/

functioning of the systems. We denotes these new event logs $Conf9$ to $Conf16$.

All the tools except CSight take event logs as input. We experimented CSight after having manually segmented $Conf1$ to $Conf8$ into trace sets, but we were unable to get any result 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 time-outs. The last steps of Csight call a model-checker to refine models with invariants, and we suspect that the model-checker was unable to check invariant satisfiability on large trace sets. Therefore, to compare CSight with the other tools, we took back two trace sets given with the CSight implementation. The first one, denoted Tcp contains 8 traces (46 events) collected from two components exchanging TCP messages. The second trace set denoted $AltBit$ contains 15 traces (246 events) expressing message exchanges between two components over the Alternating Bit Protocol, which belongs to the family of reliable transport protocols.

In summary, we considered 18 configurations. $Conf1$, 3, 5, 8, Tcp and $AltBit$ are event logs that meet the requirements of all the tools, and are particularly interesting for comparing CSight, CkTailV1, LFkbehavior and CkTailv2-w/oS. $Conf2$, 4, 6, 7 are more general event logs (composed of requests associated to multiple responses and of non-communication events) and are used to confront CkTail- v2-w/oS with LFkbehavior. Finally, $Conf9$ to 16 allow to compare our algorithms CkTailv2-w/oS and CkTailv2-w/S.

Furthermore, CkTailv1 and CkTailv2 use the procedure *ontime* to check whether an action belongs to a current session with regard to the session duration. The same procedure, which is given in Section 4.3, was used for both tools.

5.2 RQ1: can CkTailv2 infer models that capture correct behaviours of SUL?

To answer RQ1, we measured the rate of valid traces accepted by all the behavioural models generated from the 18 configurations. Given a valid trace σ and an IOLTS L, IOLTS acceptance means here that $\sigma \in Traces(L)$. To get valid traces, we chose to follow a Hold Out method, which partitioned each event log in one training log for the model generation and one testing log for the extraction of valid traces. We manually segmented event logs into two parts with an approximative ratio of 80% and 20%, taking care not to separate actions that belong to the same session to avoid the generation of incorrect models.

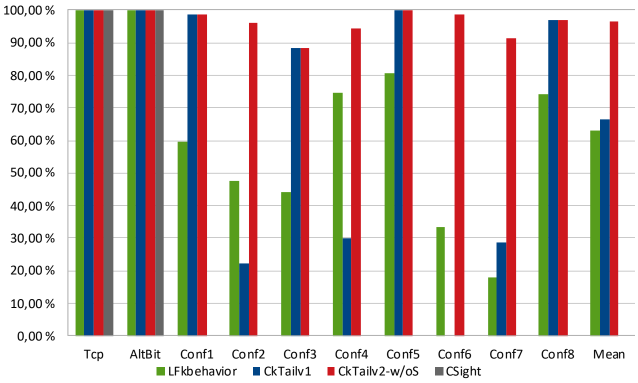

Fig. 8 Percentage of valid traces accepted by the models with the configurations Tcp, $AltBit$, $Conf1$ to 8

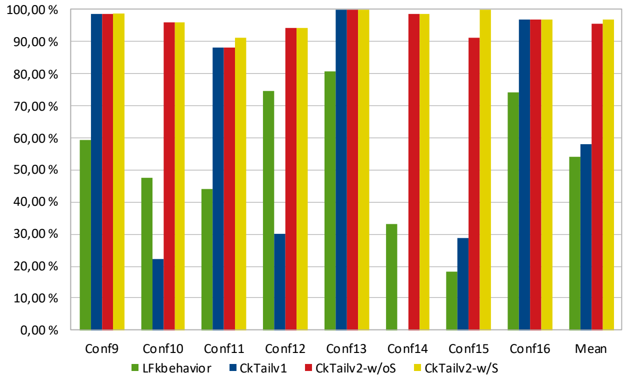

Fig. 9 Percentage of valid traces accepted by the models with the configurations $Conf9$ to 16

Afterwards, still to avoid any bias, we extracted valid trace sets from the testing logs. This trace extraction was automatically performed for the event logs including session ids. But for the other event logs, as there is no information allowing to recognise valid traces, we manually extracted them by leveraging our knowledge of the case study functioning.

We obtained around 35 to 200 valid traces for $Conf1$ to 16. For the configurations Tcp and $AltBit$ we respectively used 75 % of the traces to generate models, the remaining being used as valid traces.

Results: The percentages of valid traces accepted by the models generated by each tool are illustrated in the bar-diagrams of Figures 8 and 9. With the configurations $Conf1$ to 8, the models that accept the most of valid traces are always those generated by CkTailv2-w/oS. In our experiments, these models accept an average of 96,43% of valid traces. The models given by CkTailv1 and LFkbehavior provide close results with 66.47% and 63.23%. If we focus on the results given by CkTailv2-w/oS and CkTailV1, we have the same rate of valid traces accepted by the models with $Conf1$, 3, 5 and 8. These similarities come from the fact that these configurations meet the assumptions of both tools. The trace segmentation along with the model generation are hence performed in a similar manner. As expected, with the other configurations, we observe that CkTailv1 produced less correct models as it eliminated some actions during the trace extraction. LFkbehavior outperforms CkTailv1 with $Conf2$, 4 and 6 for the same reason.

Figure 8 also shows that the models generated from the configurations Tcp and $AltBit$ accept all the valid traces, whatever the approach used. These results tend to reveal that the sizes of the event logs used with these configurations are not large enough to make distinctions among the approaches. Therefore, we prefer to not give any conclusion here. As stated earlier, we were unable to apply CSight on larger trace sets.

Figure 9 shows that when the event logs include sessions identifiers, LFkbehavior and CkTailv1 infer models accepting the same ratio of valid traces. The interesting observation is that CkTailv2-w/oS and CkTailv2-w/S provide close results, i.e., the models given by CkTailv2-w/oS accept slightly less valid traces only. We recall that CkTailv2-w/S extracts traces from event logs by means of session identifiers (the trace extraction is always correct) whereas CkTailv2-w/oS tries to detect sessions for extracting traces. Hence, Figure 9 tends to show that the trace extraction algorithm of CkTailv2-w/oS (Algorithm 1) is very effective.

5.3 RQ2: do the models inferred by CkTailv2 reject abnormal behaviours?

This research question targets the capability of our algorithms to infer models that reject incorrect behaviours of the system. Incorrect behaviours are expressed by means of invalid traces, which are here derived from valid traces by injecting one of the following errors: repetition of actions (random addition of 2 to 6 actions), inversion of a request with its associated response(s), permutation of one request in a sequence of nested requests, and suppression of one response when several responses associated to the same request are found.

We generated 16 sets having 43 to 100 invalid traces for each configuration $Conf1$ to 16, and two sets of 20 invalid traces for Tcp and for $AltBit$. Then, we measured the proportions of invalid traces accepted by the range of models inferred from the same configurations and training sets used for RQ1.

Results: The bar-diagram of Figure 10 shows the proportions of invalid traces accepted by the models given by each tool in each configuration. This figure reveals that all the tools performed well in the sense that the inferred models reject most of the incorrect behaviours. CkTailv1, CkTailv2 and CSight outperform LFkbehavior with some configurations though. For instance, LFkbehavior produced models that accept 13,3% of invalid traces with $Conf5$.

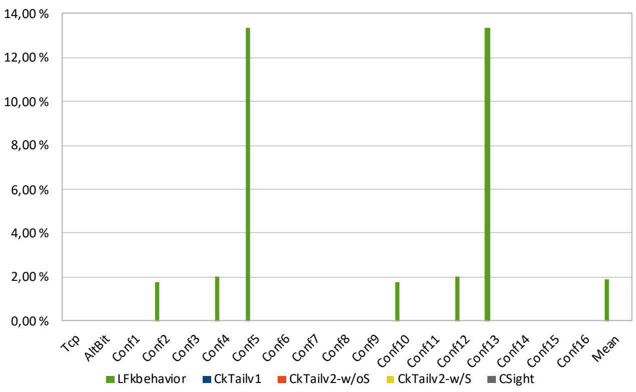

Fig. 10 Percentage of invalid traces accepted by the models for each configuration

As previously, it remains difficult to compare CSight and CkTailv2 because only two configurations Tcp and $AltBit$ are considered in this evaluation. As CSight uses invariant satisfiability to increase the model precision and not CkTailv2, we believe that CSight should return more precise models, but only with small trace sets.

The results given with RQ1 and RQ2 tend to indicate that the models produced by CkTailv2 offer the best precision: not only they accept the highest ratio of valid traces, but also reject all the invalid ones (as CSight).

5.4 RQ3: can CkTailv2 detect accurate dependencies among components?

This research question investigates the capability of our algorithms to find component dependencies during the event log analysis. Among the range of tools considered in this evaluation, only CkTailv1, CkTailv2-w/oS and CkTailv2-w/S are able to infer dependency graphs, but CkTailv1 and CkTail- v2-w/oS use the same dependency detection. As a consequence, we chose to study RQ3 by comparing the DAGs returned by CkTailv2-w/oS and CkTailv2-w/S to the real dependency graphs we manually built from the dependency schemes that we devised for $Conf1$ to 8, Tcp and $AltBit$. We evaluated the recall and precision of both algorithms. A good component dependency detection is characterised by a high recall and high precision, where high recall also relates to a low false negative rate and high precision relates to low false positive rate.

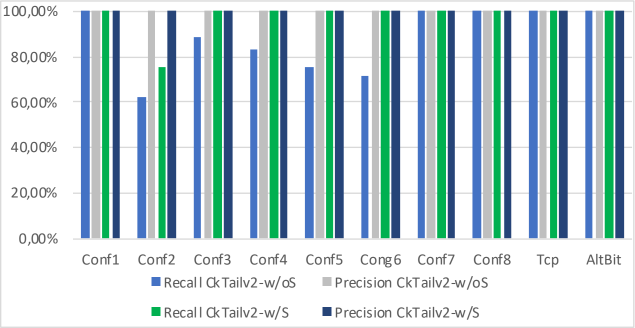

Fig. 11 recall and precision of CkTailv2 to detect component dependencies. Recall is the percentage of the real dependencies that are detected; precision is the percentage of detected dependencies that are correct

$Conf1$	$Conf2$	$Conf3$	$Conf4$	$Conf5$	$Conf6$	$Conf7$	$Conf8$	Tcp	$AltBit$
8	8	9	6	8	7	4	10	2	2

Table 3 # real dependencies for each configuration

Results: Table 3 shows the number of real component dependencies for each configuration and the bar-diagram of Figure 11 depicts the recalls and precisions achieved by CkTail- v2-w/oS and CkTailv2-w/S. On average, CkTailv2-w/oS detected 88% of the real dependencies and CkTailv2-w/S 97,5%. No wrong component dependency is returned by both algorithms. After inspection, we observed that the undiscovered dependencies correspond to some data dependencies that can be observed among several chains of messages sharing the same data addressed to several components at the same time. We have chosen in Definition 3 to not consider them to avoid returning false dependencies. This case of having chains of messages sharing the same data addressed to several components is more frequent with CkTailv2-w/oS as it detects data dependencies on the action sequence S, while CkTailv2-w/S does it on traces, which are smaller sequences. As a consequence, the recall of CkTailv2-w/oS is lower than the one of CkTailv2-w/S.

5.5 RQ4: what is the performance of CkTailv2 to infer models compared to the other tools? How does CkTailv2 scale with the size of the event log?

Procedure To answer RQ4, we firstly studied how the tools scale with the size of the event logs. We collected 40 event logs from $Conf3$ by varying the number of events between 500 to 10000 events. Then, we measured execution times to produce models. As CSight did not complete on $Conf3$, we only considered LFkbehavior, CkTailv1, CkTailv2-w/oS and CkTailv2-w/S. To include CSight in our evaluation, we measured the execution times of all the tools on Tcp and $AltBit$. Experiments were carried out on a computer with 1 Intel® CPU i5-6500 @ 3.2GHz and 32GB RAM.

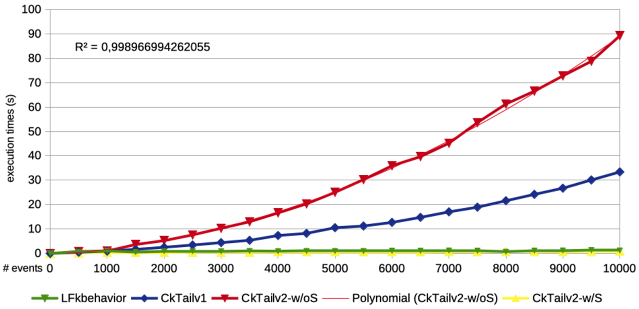

Fig. 12 Execution times vs. number of events

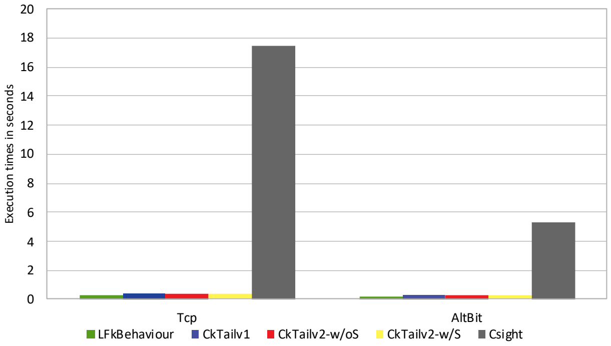

Fig. 13 Execution times of the tools with the configurations Tcp and $AltBit$

Results Figure 12 depicts the execution times in seconds of the tools w.r.t. the event log sizes. CkTailv2-w/S offers the best performance as it produces models in less than 1 second. These results are not surprising as the algorithm splits event logs quickly thanks to the session identifiers. LFkbehavior offers close results as it never took more than 2 seconds to produce models. On the other hand, CkTailv1 and CkTailv2-w/oS required less than 33s and 89s respectively. The curve for CkTailv2-w/oS follows a quadratic regression and reveals that our tool does not scale well. Most of the execution times are consumed by the trace extraction step to recover sessions, one after the other. The difference between CkTailv1 and CkTailv2-w/oS comes from the fact that CkTailv2-w/oS uses two set of pending requests to check if the constraints C1-C5 hold while CkTailv1 needs one set only.

The bar-diagram of Figure 13 illustrates the execution times of all the tools on the configurations Tcp and $AltBit$. These results tend to show that CSight is significantly slower than the other tools, it is around 30 times slower than CkTail- v2-w/oS. Besides, as stated earlier, CSight were unable to return models after 5 hours with $Conf1$ to 8.

These experiments show that CkTailv2 can be used in practice to infer models of communicating systems even with large event logs, but it suffers from insufficient scalability, on account of its feature of detecting sessions for extracting traces.

5.6 Threat to Validity

Some threats to validity can be identified in our evaluation. The first factor, which may threaten the external validity of our results, applies to the case studies used in the experimentations. Most of them indeed are IoT systems using the HTTP protocol. We also considered two other event logs collected from components exchanging messages by means of the TCP and Alternating bit protocols. But many communicating systems rely on other kinds of protocols, from which it may be more difficult to identify senders, receivers, requests or responses. Hence, our results cannot be generalised to all communicating systems. This is why we deliberately avoid drawing any general conclusion. We chose to mainly concentrate our experimentations on IoT systems that we devised to be able to appraise the capability of CkTailv2 of inferring correct dependency graphs. This threat is somewhat mitigated by the fact that our results can be easily generalised to communicating systems based upon the HTTP protocol, and that the latter is used by numerous communicating systems.

The generalisation of our approach is also restricted by the requirements A1-A3. The event logs have to include timestamps given by a global clock and must be formatted by means of regular expressions so that the event types can be identified. Although we have observed that this task is not too difficult to carry out on HTTP messages, it is manifest that this is not generalisable to any kind of protocols, especially those encrypting some parts of the message contents. We need to investigate how these requirements could be relaxed in future work.

There are also some threats to internal validity. Firstly, like all the other passive model learning approaches, the larger the event log, the more complete and precise the models will be. Furthermore, our approach uses one parameter denoted T, in the procedure *ontime*, to limit the session duration. We set this parameter to 1 or 2 seconds in our experiments as the session durations was lower than these values in our case studies. Changing this parameter impacts the precision of the models though. We assume that the user has some knowledge about SUL and that he/she can set this parameter correctly. Otherwise, we suggest to generate several models while modifying this parameter. We evaluated the precisions of the models generated from $Conf2$ with T taking values between 0 and 150 seconds. Figure 14 illustrates the ratios of valid and invalid traces accepted by the inferred models. The ratio of invalid traces remain unchanged. But, the ratio of valid traces evolves with T. Although the figure does not allow to directly deduce the best parameter value as several ones are possible, it helps avoid choosing the bad ones.

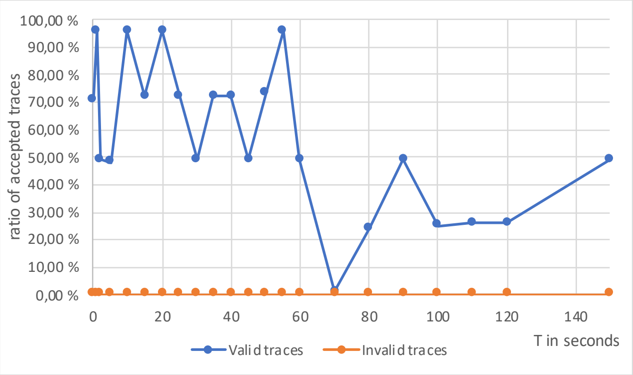

Fig. 14 Impact of the session duration on the model precision

6 Conclusion

This paper has proposed the design and evaluation of a tool called CkTailv2, which is specialised into the learning of behavioural models along with dependency graphs from event logs, themselves collected from of communicating systems. Compared to other model learning algorithms, CkTailv2 increases the precision of the generated models by integrating an algorithm that better recognises sessions in event logs with respect to constraints related to the request-response pattern, the recognition of nested requests, time delays and component dependency. Besides, when sessions are explicitly identified in event logs, CkTailv2 provides another algorithm to quicker generate models.

CkTailv2 is simple to use. A user only has to give an event log and a set of regular expressions as inputs to produce one IOLTS and one DAG per component of the communicating system. These models are stored in DOT files and varied tools can process them to graphically represent how the communicating system behaves and is structured. These models may be then used to detect defects or security vulnerabilities. Besides, our evaluation showed that CkTailv2 is effective, as it provides precise models, and that it can be used in practice on large event logs.

Nevertheless, several aspects need to be investigated and improved in the future. We firstly plan to evaluate CkTailv2 on further kinds of systems to confirm our experimental results. The latter show that CkTailv2 does not scale well with the size of the event logs. We believe that the performance can be improved by devising parallel algorithms. But another way is to get rid of some requirements, such as the need to have events that encode senders and receivers. We believe that an additional event log analysis step could perform this task automatically.

Another direction of future work is to make use of these models to assist developers in the analysis and test of communicating systems. More precisely, we intend to propose an approach combining this model learning technique with the generation of mocks, i.e. fake components that simulate real components and that behave in a predefined way. These mock components could make test development easier by replacing complex dependencies (e.g., infrastructure or environment related dependencies [6, 40]). Besides, mock components could increase test efficiency by replacing slow-to-access components. We finally believe that the models produced by CkTailv2 could be analysed to automatically generate executable mock components.

7 Acknowledgement

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

References

1. Ahmad A, Bouquet F, Fourneret E, Le Gall F, Legeard B (2016) Model-based testing as a service for iot platforms. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications, Springer International Publishing, Cham, pp 727–742
2. Aichernig BK, Tappler M (2017) Learning from faults: Mutation testing in active automata learning - mutation testing in active automata learning. In: NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings, pp 19–34, DOI 10.1007/978-3-319-57288-8_2
3. Alur R, Černý P, Madhusudan P, Nam W (2005) Synthesis of interface specifications for java classes. SIGPLAN Not 40(1):98–109, DOI 10.1145/1047659.1040314
4. Ammons G, Bodík R, Larus JR (2002) Mining specifications. SIGPLAN Not 37(1):4–16, DOI 10.1145/565816.503275
5. Angluin D (1987) Learning regular sets from queries and counterexamples. Information and Computation 75(2):87 – 106
6. Arcuri A, Fraser G, Just R (2017) Private api access and functional mocking in automated unit test generation. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp 126–137
7. Berg T, Jonsson B, Raffelt H (2006) Regular inference for state machines with parameters. In: Baresi L, Heckel R (eds) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 3922, Springer Berlin Heidelberg, pp 107–121, DOI 10.1007/11693017_10
8. Beschastnikh I, Brun Y, Schneider S, Sloan M, Ernst MD (2011) Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE '11, pp 267–277
9. Beschastnikh I, Brun Y, Ernst MD, 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, ACM, New York, NY, USA, ICSE 2014, pp 468–479, DOI 10.1145/2568225.2568246, URL http://doi.acm.org/10.1145/2568225.2568246
10. Biermann A, Feldman J (1972) On the synthesis of finite-state machines from samples of their behavior. Computers, IEEE Transactions on C-21(6):592–597, DOI 10.1109/TC.1972.5009015
11. van der Bijl M, Rensink A, Tretmans J (2004) Compositional testing with ioco. In: Petrenko A, Ulrich A (eds) Formal Approaches to Software Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 86–100
12. Blot E, Salva S (2020) The cktailv2 tool. URL https://github.com/sasa27/CkTailv2
13. Blot E, Salva S (2020) Testability measurments on inffered models. URL https://github.com/Elblot/testability
14. Dssouli R, Karoui K, Petrenko A, Rafiq O (1996) Towards testable communication software. In: Cavalli A, Budkowski S (eds) Protocol Test Systems VIII: Proceedings of the IFIP WG6.1 TC6 Eighth International Workshop on Protocol Test Systems, September 1995, Springer US, Boston, MA, pp 237–251, DOI 10.1007/978-0-387-34988-6_15

15. Dupont P (1996) Incremental regular inference. In: Proceedings of the Third ICGI-96, Springer, pp 222–237
16. ETSI (2015) Methods for testing & specification; risk-based security assessment and testing methodologies, european telecommunications standards institute, technical report, https://www.etsi.org/deliver/etsi_eg/203200_203299/203251/01.01.01_50/eg_203251v010101m.pdf
17. Falcone Y, Jaber M, Nguyen TH, Bozga M, Bensalem S (2011) Runtime Verification of Component-Based Systems. In: Barthe G, Pardo A, Schneider G (eds) SEFM 2011 - Proceedings of the 9th International Conference on Software Engineering and Formal Methods, Springer, Montevideo, Uruguay, Lecture Notes in Computer Science (LNCS), vol 7041, pp 204–220, DOI 10.1007/978-3-642-24690-6_15, URL https://hal.archives-ouvertes.fr/hal-00642969
18. Fu Q, Lou JG, Wang Y, Li J (2009) Execution anomaly detection in distributed systems through unstructured log analysis. 2009 Ninth IEEE International Conference on Data Mining pp 149–158
19. Groz R, Li K, Petrenko A, Shahbaz M (2008) Modular system verification by inference, testing and reachability analysis. In: Suzuki K, Higashino T, Ulrich A, Hasegawa T (eds) Testing of Software and Communicating Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 216–233
20. Gutiérrez-Madroñal L, Medina-Bulo I, Domínguez-Jiménez J (2018) Iot–teg: Test event generator system. Journal of Systems and Software 137:784 – 803, DOI https://doi.org/10.1016/j.jss.2017.06.037, URL http://www.sciencedirect.com/science/article/pii/S0164121217301280
21. Hossen K, Groz R, Oriat C, Richier J (2014) Automatic model inference of web applications for security testing. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA, pp 22–23, DOI 10.1109/ICSTW.2014.47
22. Howar F, Steffen B, Jonsson B, Cassel S (2012) Inferring canonical register automata. In: Kuncak V, Rybalchenko A (eds) Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol 7148, Springer Berlin Heidelberg, pp 251–266, DOI 10.1007/978-3-642-27940-9_17
23. Lo D, Mariani L, Santoro M (2012) Learning extended fsa from software: An empirical assessment. Journal of Systems and Software 85(9):2063 – 2076, DOI http://dx.doi.org/10.1016/j.jss.2012.04.001, URL http://www.sciencedirect.com/science/article/pii/S0164121212001008, selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011)
24. Lorenzoli D, Mariani L, Pezzè M (2008) Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE'08, pp 501–510
25. Makanju A, Zincir-Heywood AN, Milios EE (2012) A lightweight algorithm for message type extraction in system application logs. IEEE Transactions on Knowledge and Data Engineering 24(11):1921–1936, DOI 10.1109/TKDE.2011.138
26. Mariani L, Pastore F (2008) Automated identification of failure causes in system logs. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on, pp 117–126, DOI 10.1109/ISSRE.2008.48
27. Mariani L, Pezze M (2007) Dynamic detection of cots component incompatibility. IEEE Software 24(5):76–85, DOI http://doi.ieeecomputersociety.org/10.1109/MS.2007.138
28. Mariani L, Pezzè M, Santoro M (2017) Gk-tail+ an efficient approach to learn software models. IEEE Transactions on Software Engineering 43(8):715–738, DOI 10.1109/TSE.2016.2623623
29. Messaoudi S, Panichella A, Bianculli D, Briand L, Sasnauskas R (2018) A search-based approach for accurate identification of log message formats. In: Proceedings of the 26th Conference on Program Comprehension, ACM, New York, NY, USA, ICPC '18, pp 167–177, DOI 10.1145/3196321.3196340, URL http://doi.acm.org/10.1145/3196321.3196340
30. Ohmann T, Herzberg M, Fiss S, Halbert A, Palyart M, Beschastnikh I, Brun Y (2014) Behavioral resource-aware model inference. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE '14, pp 19–30
31. Pastore F, Micucci D, Mariani L (2017) Timed k-tail: Automatic inference of timed automata. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp 401–411, DOI 10.1109/ICST.2017.43
32. Petrenko A, Avellaneda F (2019) Learning communicating state machines. In: Beyer D, Keller C (eds) 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, Springer, Lecture Notes in Computer Science, vol 11823, pp 112–128, DOI 10.1007/978-3-030-31157-5, URL https://doi.org/10.1007/978-3-030-31157-5

33. Raffelt H, Steffen B, Berg T (2005) Learnlib: A library for automata learning and experimentation. In: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, ACM, New York, NY, USA, FMICS '05, pp 62–71, DOI 10.1145/1081180.1081189
34. Salva S, Blot E (2019) Reverse engineering behavioural models of iot devices. In: 31st International Conference on Software Engineering & Knowledge Engineering (SEKE), Lisbon, Portugal, URL https://hal-clermont-univ.archives-ouvertes.fr/hal-02134046
35. Salva S, Blot E (2019) Reverse engineering behavioural models of iot devices. In: 31st International Conference on Software Engineering & Knowledge Engineering (SEKE), Lisbon, Portugal, URL https://hal-clermont-univ.archives-ouvertes.fr/hal-02134046
36. Salva S, Blot E (2020) Cktail: Model learning of communicating systems. In: Ali R, Kaindl H, Maciaszek LA (eds) Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020, SCITEPRESS, pp 27–38, DOI 10.5220/0009327400270038, URL https://doi.org/10.5220/0009327400270038
37. Salva S, Blot E (2020) Model generation of component-based systems. Software Quality Journal 28(2):789–819, DOI 10.1007/s11219-019-09485-y
38. Salva S, Blot E (2020) Verifying the application of security measures in iot software systems with model learning. In: Proceedings of the 15th 15th International Conference on Software Technologies, ICSOFT 2020, Paris, France, july, 2020, pp 1–12
39. Salva S, Fouchal H, Bloch S (2000) Metrics for timed systems testing. In: Procedings of the 4th International Conference on Principles of Distributed Systems, OPODIS 2000, Paris, France, December 20-22, 2000, pp 177–200
40. Spadini D, Aniche M, Bruntink M, Bacchelli A (2019) Mock objects for testing java systems. Empirical Softw Engg 24(3):1461–1498, DOI 10.1007/s10664-018-9663-0, URL https://doi.org/10.1007/s10664-018-9663-0
41. Tappler M, Aichernig BK, Bloem R (2017) Model-based testing iot communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp 276–287, DOI 10.1109/ICST.2017.32
42. Vaarandi R, 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), pp 1–7, DOI 10.1109/CNSM.2015.7367331
43. Zhu J, He S, Liu J, He P, Xie Q, Zheng Z, Lyu MR (2018) Tools and benchmarks for automated log parsing. CoRR abs/1811.03509, URL http://arxiv.org/abs/1811.03509, 1811.03509