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

# Model Generation of Component-based Systems.

**Sébastien Salva*** · **Elliott Blot**

the date of receipt and acceptance should be inserted later

**Abstract** This paper presents COnfECt, a model learning approach, which aims at recovering the functioning of a component-based system from its execution traces. We refer here to non concurrent systems whose internal interactions among components are not observable from the environment. COnfECt is specialised into the detection of components of a black-box system and in the inference of models called systems of Labelled Transition Systems (LTS). COnfECt tries to detect components and their specific behaviours in traces, then it generates a LTS for every component discovered, which captures its behaviours. Besides, it synchronises the LTSs together to express the functioning of the whole system. COnfECt relies on machine learning techniques to build models: it uses the notion of correlation among actions in traces to detect component behaviours, and exploits a clustering technique to merge similar LTSs and synchronise them. We describe the three steps of COnfECt and the related algorithms in this paper. Then, we present some preliminary experimentations.

**Keywords** Model Learning; Formal models; Reverse Engineering; Component-based Systems.

## 1 Introduction

The effort required for writing (formal) models has been a strong barrier to the widespread adoption of model-based testing or verification approaches in the industry. Most of today's developers indeed feel that writing models is a difficult, long and error-prone task. This obstacle can be overcome with model learning approaches (Angluin, 1987; Biermann and Feldman, 1972; Ernst et al, 1999; Meinke

Sébastien Salva
* Corresponding author
University Clermont Auvergne, IUT of Clermont-Ferrand, LIMOS, F-63000 CLERMONT-FERRAND, FRANCE
E-mail: sebastien.salva@uca.fr

Elliott Blot
University Clermont Auvergne, LIMOS, F-63000 CLERMONT-FERRAND, FRANCE
E-mail: elliott.blot@uca.fr

and Sindhu, 2011; Lorenzoli et al, 2008; Ohmann et al, 2014; Durand and Salva, 2015; Pastore et al, 2017), which have proven to be valuable for recovering models that can be exploited in several software engineering steps. The inferred models can be seen as documentation useful for understanding the functioning of a system; they can be completed and improved to become formal specifications; several papers also showed that model learning can be employed within effective bug finding techniques (Mariani and Pastore, 2008; Hangal and Lam, 2002; Tappler et al, 2017), or can be used to directly generate test cases (Dallmeier et al, 2012; Shahbaz and Groz, 2013; Durand and Salva, 2015).

A substantial part of these approaches are specialised to infer models from black-box systems. These offer the advantage of remaining usable when the application code is not available or when the system internal state cannot be known. These approaches capture the behaviours of systems interacting with their environments in a variety of models, e.g., states machines (Angluin, 1987; Biermann and Feldman, 1972) or invariants (Ernst et al, 1999; Meinke and Sindhu, 2011). The model generation is performed either by interacting with the system (active approaches), or by analysing a set of execution traces resulting from the monitoring of the system (passive approaches). In this paper, we focus on the second category.

In this context, several papers recently proposed innovative solutions for designing new learning algorithms and tools, which have the capability to infer symbolic models (Mariani et al, 2017), resource-aware models (Beschastnikh et al, 2011; Ohmann et al, 2014), timed models (Pastore et al, 2017), and which can be applied to more and more complex systems. But, few works (Groz et al, 2008; Mariani and Pastore, 2008; Beschastnikh et al, 2014) dealt with the generation of models from integrated systems made up of components. Yet numerous present systems are constituted of reusable features or components, which interact together. The inference of models encoding the functioning of every component into a sub-model and how they interact together would greatly ease the readability and analysis of the whole system. Furthermore, such models would offer the possibility to concentrate the efforts for bug detection on some specific sub-parts of the system. These observations motivate this work, which addresses these two research challenges: **Challenge 1:** given a system under learning SUL, how to learn a model from its execution traces, in such a way that the model captures the behaviours of the SUL components and their synchronisations? **Challenge 2:** how to manage the level of generalisation of the models, and how to synchronise the sub-models of components?

To address these challenges, we designed a new method called COnfECt (COrrelate Extract Compose) and a corresponding tool for learning models of component-based systems. COnfECt is a passive model learning approach, which generates a system of LTSs (Labelled Transition Systems) from execution traces of non concurrent systems whose internal interactions among components are not observable from the environment. This is often the case for systems having a tightly coupled architecture (strong dependency among components), e.g., embedded systems (vending machines, electronic toys), Internet Of Things (IoT) devices (smart thermostat, security camera) or for software made up of modules or subprograms that are dependent upon each other.

COnfECt is composed of three main steps called *Trace Recovery*, *Trace Analysis & Extraction* and *LTS synchronisation*. The first step derives formatted traces

from raw messages, the second analyses the traces, tries to identify distinctive sub-sequences in traces and to link them to separate components. The last step generates a system of LTSs by means of three strategies. This model encodes the behaviours of every component by a LTS and shows how they are synchronised together. The strategies adapt the LTS synchronisation, and provide several systems of LTSs having different levels of generalisation. These steps rely on machine learning techniques to detect the behaviours of components: traces are analysed with Correlation factors based on String similarity metrics and algorithms; the LTS Synchronisation step relies on a clustering technique to group similar LTSs.

We have implemented a prototype tool to experiment COnfECt and evaluate its benefits. We provide a preliminary evaluation in the paper, which assesses the correct component detection, the relevance and size of the models, and the efficiency/scalability of the algorithm, compared to two other model learning approaches kTail and CSight. We also examine potential threats to the validity of our evaluation.

**Paper organisation:** Section 2 presents some papers related to our approach and our motivations. Section 3 recalls some definitions about the LTS model. Section 4 gives an overview of the functioning of COnfECt with an example. Section 5 describes the steps of the approach. The next section shows the results of the experimentation of COnfECt and discusses about the threats to validity. Finally, Section 8 summarises our contributions and draws some perspectives for future work.

## 2 Related Work

Model learning can be defined as *a set of methods that infer a specification by gathering and analysing system executions and concisely summarising the frequent interaction patterns as state machines that capture the system behaviour* (Ammons et al, 2002). These models, even if partial, can serve many purposes, e.g., they can be used as documentation, examined by designers to find bugs, or can be given to testing methods for the test case generation. Models can be generated from different kinds of data samples such as affirmative/negative answers (Angluin, 1987), execution traces (Krka et al, 2010), source code (Pradel and Gross, 2009), or network traces (Antunes et al, 2011).

Most of the approaches fall into two categories called active and passive model learning, although some works cover both (Petrenko et al, 2017). Active learning approaches, e.g., (Angluin, 1987; Dupont, 1996; Raffelt et al, 2005; Alur et al, 2005; Berg et al, 2006; Howar et al, 2012; Hossen et al, 2014), repeatedly query systems or humans to collect positive or negative observations, which are studied to build models. Many existing active techniques have been conceived upon two concepts, the $\mathcal{L}^*$ algorithm (Angluin, 1987) and incremental learning (Dupont, 1996). This model learning category is actively studied to make the approaches more effective and efficient. For instance, some researchers recently proposed optimisations to reduce the query number (Aichernig and Tappler, 2017), while others tackled systems having specific constraints (Hossen et al, 2014). Active learning cannot be applied on any system though. For instance, uncontrollable systems cannot be queried easily, or the use of active testing techniques may lead a system to abnormal functioning, because it has to be reset many times.

This brings us to the second category, which includes the techniques that passively generate models from a given set of samples, e.g., a set of execution traces. These techniques are said passive since there is no direct interaction with the system. Models are here often constructed by encoding sample sets with automata whose equivalent states are merged. The state equivalence is usually defined by means of state-based abstractions or event sequence abstractions. The approaches that use state-based abstractions, e.g., (Meinke and Sindhu, 2011), adopted the generation of state-based invariants to define equivalence classes of states that are combined together to form final models. The Daikon tool (Ernst et al, 1999) was originally proposed to infer invariants composed of data values and variables found in execution traces. With event sequence abstractions, the abstraction level of the models is raised by merging equivalent states (Biermann and Feldman, 1972; Mariani and Pezze, 2007). In the kTail approach (Biermann and Feldman, 1972), the equivalent states are those having the same k-future, i.e. the same event sequences having the maximum length $k$. kTail has been later enhanced with Gk-tail to generate Extended Finite State Machines encoding data constraints (Lorenzoli et al, 2008; Mariani et al, 2017). The methods Synoptic (Beschastnikh et al, 2011) and Perfume (Ohmann et al, 2014) also reuse kTail. The former generates more precise models by means of the generation of temporal invariants from logs, which have to be satisfied by the models. The later, which is an improvement of Synoptic, infers resource-aware models capturing behavioural executions that differ in resource consumption. More recently, Pastore et al (2017) proposed Tk-tail to support the learning of timed automata.

## 2.1 Key observations and motivations

After having studied the literature, we have firstly observed that few papers tackled Challenge 1 or 2. Groz et al (2008) proposed to generate a controllable approximation of components through active testing. Unlike our approach, the learning of the components is done in isolation, i.e. there is no detection of components as these are known and studied one after the other. Mariani and Pastore (2008) proposed an automatic detection of failures in log files by means of model learning. Their approach offers the possibility to split the original log file into as many files as components. The latter are distinguished in logs by means of regular expressions, which have to be written by end-users. Once the trace set is segmented by component, the models are generated in isolation. Unlike the LTSs provided by COnfECt, these models do not show how components are synchronised.

CSight (Beschastnikh et al, 2014) seems to be the major approach that shares several purposes of COnfECt. Csight infers models of concurrent communicating systems, which communicate through synchronous channels. It is assumed that the channels and components are known. Csight also requires specific trace sets capturing this notion of channels: the trace set is segmented with one subset (called process trace set) by component. The exchanged messages are observable and composed of input and output events. Csight has five main stages: 1) log parsing and mining of invariants that must hold in the models 2) generation of a concrete FSM that captures the functioning of the whole system by recomposing the traces of the different components; 2) generation of a more concise abstract FSM; 3) model refinement with invariants, and 4) generation of Communicating FSM (CFSM).

The latter show the synchronizations of the concurrent components by means of the channels and of the input/ outputs, e.g., when the emitter sends an output, the receiver gets an input with the same symbol and vice versa. COnfECt aims at learning models from traces of component-based systems, where the component interactions are hidden. In the paper, we will use as example a connected thermostat integrating several components. With this king of system, the component interaction is not observable. As a consequence, CSight cannot infer a model per component, whereas COnfECt decomposes traces to recover the component behaviours and infer models. Furthermore, we consider that the component number is unknown and traces are not segmented. Hence the assumptions required by CSight and COnfECt are quite different. But COnfECt needs of other assumptions on the system under learning. In a way, COnfECt targets more the systems having a tightly coupled architecture, whereas CSight seems more to target the systems having a loosely-coupled architecture, where components can remain autonomous and allow middleware software to manage communication between them. The models given by CSight should be more precise than those given by COnfECt because CFMS have to be compliant with the behaviours of the traces (thanks to the mining and satisfiability of invariants). At the moment, we do not focus on the satisfiability of mined invariants as this topic has been studied in several papers, e.g., (Beschastnikh et al, 2014; Ohmann et al, 2014). However, this could be implemented in COnfECt in future work. Instead, our approach proposes three strategies to adapt the model generalisation (from a model that is compliant to the behaviours found in traces to a more general model that may call its related components at each of its states). We believe that this notion of generalisation level is important as the original traces may only capture a part of the real behaviours of a system.

Prior to this paper, we laid the first stone of the approach in (Salva and Blot, 2018), in which we proposed to complement Gk-tail for the generation of models of component-based systems. We defined the CEFSM model (Callable Extended Finite State Machine), which is composed of variables and constraints. CEFSMs cannot be composed together though, which reduces their re-usability. Besides, we had not implemented the given algorithms nor evaluated them. We also proposed an overview of this work in (Salva et al, 2018). Like in this paper, we considered the LTS model so that we can reuse the LTS theoretical background. We introduced the general functioning of COnfECt and started an evaluation on the component detection. In this paper, we define the Correlation coefficient allowing to recognise the call of components in traces. We define the LTS similarity coefficient allowing to provide several LTS synchronisation strategies. Furthermore, we present the algorithms implementing the steps of COnfECt and provide the results of a more thorough evaluation carried out to assess the relevance of the models generated by COnfECt and its efficiency.

## 3 Preliminary Definitions

COnfECt aims to generate models of component-based systems, where the interactions among components are not observable. The specific notions of communications or channels, which can be found in specific models such as the CFSM, are not required here. Like in (Falcone et al, 2011; van der Bijl et al, 2004), we propose

to express the behaviours of atomic components with the well established Labelled Transition System (LTS) model. The use of LTS allows to exploit the definitions related to the LTS composition, for instance given by van der Bijl et al (2004). A composite model, which we denote system of LTSs, is defined with respect to the LTS parallel composition, which synchronises LTSs on their shared actions, called *synchronisation actions.*

The LTS model is firstly defined in terms of states and transitions labelled by actions, taken from a general action set $\mathcal{L}$, which expresses what happens. $\tau$ is a special symbol encoding an internal (unobservable) action; it is common to denote the set $\mathcal{L} \cup \tau$ by $\mathcal{L}_\tau$.

**Definition 1 (LTS)** A Labelled Transition System (LTS) is a 4-tuple $\langle Q, q0, \Sigma, \rightarrow\rangle$ where :

- $Q$ is a finite set of states, $Q_F \subseteq Q$ is the non-empty set of final states;
- $q0$ is the initial state;
- $\Sigma \cup \{\tau\} \subseteq \mathcal{L}_\tau$ is the finite set of actions, with $\tau$ the internal action;
- $\rightarrow \subseteq Q \times \Sigma \cup \{\tau\} \times Q$ is a finite set of transitions. A transition $(q, a, q')$ is also denoted $q \xrightarrow{a} q'$.

We use the generalised transition relation $\rightarrow$ to represent LTS paths: $q \xrightarrow{a_1 \dots a_n} q' =_{def} \exists q0 \dots q_n, q = q0 \xrightarrow{a_1} q_1 \dots q_{n-1} \xrightarrow{a_n} q_n = q'$. The concatenation of two action sequences $\sigma_1$, $\sigma_2 \in \mathcal{L}_\tau^*$ is denoted $\sigma_1.\sigma_2$. $\epsilon$ denotes the empty sequence. Finally, we define the runs and traces of a LTS:

**Definition 2 (Runs and traces)** Let $L = \langle Q, q0, \Sigma, \rightarrow\rangle$ be a LTS.

1. A run $q0a_1...q_{k-1}a_kq_k$ is an alternate sequence of states and actions such that: $\exists q_{i-1}, q_i, a_i, (1 \leq i \leq k) : q0 \xrightarrow{a_1 \dots a_k} q_k \in \rightarrow^*$. $Runs(L)$ is the set of runs found in $L$. $Runs_F(L)$ is the set of runs that end in a state $q$ of $F$ with $F \subseteq Q$;
2. the trace of a run $r = q0a_1...q_{k-1}a_kq_k$, denoted $Trace(r)$ is the sequence $a_0...a_k$. $Traces_F(L) = \{Trace(r) \mid r \in Runs_F(L)\}$;

The integration of two components $C_1$ and $C_2$ modelled with LTSs is often defined in the literature by two operations. The first one is the parallel composition of $C_1$ and $C_2$ denoted $C_1 \parallel C_2$, which synchronises their synchronisation actions. This composition is often followed by the hiding of the communications between $C_1$ and $C_2$ to express that only the communications with the environment are observable. This operation is defined by the relation *hide* $S$ in $C_1 \parallel C_2$ with $S$ the set of synchronisation actions. We refer to (van der Bijl et al, 2004) for the definitions of theses two LTS operators. This principle of LTS composition leads to a model called system of LTSs, which describes a component-based system:

**Definition 3 (System of LTSs)** A system of LTSs $SC$ is the couple $\langle S, C\rangle$ with $C = \{C_1, \dots, C_n\}$ a non empty set of LTSs, and $S$ a set of synchronisation actions.

$Traces(SC)$ denotes the trace set $Traces((hide\ S\ in\ (C_1 \parallel C_2 \parallel \cdots \parallel C_n))$.

## 4 COnfECt Overview

COnfECt (COrrelate Extract Compose) firstly aims at answering to Challenge 1: how to infer a system of LTSs $SC$ from the traces of SUL, in such a way that

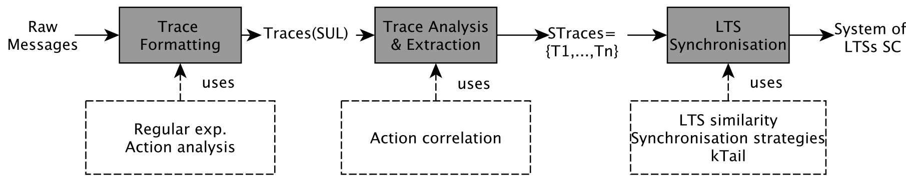



Fig. 1: The COnfECt approach overview

*SC* captures the behaviours of the SUL components and their synchronisations? Initially, COnfECt requires a set of raw messages collected from SUL. The latter can be non-deterministic, uncontrollable or can have cycles among its internal states. However, to answer to this research problem, we assume that SUL obeys certain restrictions:

- **H1:** SUL **as a black box.** SUL is a black box including components from which only communications with the environment can be observed. The interactions among the components are not observable.
- **H2: Synchronous execution.** SUL has components whose behaviours are not carried out in parallel. One component is executed at a time from its initial state to one of its final states. Furthermore, we assume that messages include timestamps.
- **H3: Single root component.** We suppose that the traces of $Traces$(SUL) capture the behaviours of one first component calling other components.

Regarding the assumptions H1-H3, we assume that components follow a procedural behaviour: a component C1 calls another component C2 and waits for the end of its execution. C2 starts its execution at its initial state and does some actions. Once the execution of C2 has completed, C1 proceeds its execution. These assumptions are considered to lay the foundations of a component detection algorithm from traces. These are discussed in Section 7.5. Relaxing these assumptions remains as part of future work. COnfECt has three main successive steps illustrated in Figure 1. The first step takes raw messages given by monitoring tools or found in log files, and transforms them into formatted traces. The second step, called *Trace Analysis & Extraction*, tries to detect component behaviours in $Traces$(SUL), and partitions it into a set of trace sets called $STraces$. Each trace set of $STraces$ captures some behaviours of one component. The last step, called *LTS Synchronisation*, takes $STraces$ and starts with the generation of one LTS for each trace set of $STraces$. This step also proposes three LTS synchronisation strategies to generate a system of LTSs $SC$.

Figure 3 illustrates an overview of these steps with an example of system, which is a connected thermostat device from which HTTP traces may be collected. The user gives as inputs: a file of messages, regular expressions to build traces, 2 factors ($f$ $f'$) allowing to express similarities among actions and components, two thresholds for these factors, and a LTS synchronisation strategy. Figure 2 lists 4 HTTP messages collected from the device taken as example. COnfECt starts by parsing these messages with regular expressions to produce the set $Traces$(SUL). An example of regular expression is also given in Figure 2. It extracts parameters and a label, which shows what happens.

```
Jul 18, 2018 08:52:26.696766000 CET;Host=192.168.1.44;Dest=192.168.
1.4;Protocol=HTTP;Verb=GET Uri=/devices HTTP/1.1;
Jul 18, 2018 08:52:30.362482000 CET;Host=192.168.1.4;Dest=192.168.
1.44;Protocol=HTTP;Verb=GET Uri=/json.htm?type=command&param=
udevice&idx=115&nvalue=0&svalue=15.00 HTTP/1.1;
Jul 18, 2018 08:52:30.522163000 CET;Host=192.168.1.44;Dest=192.168.
1.4;Protocol=HTTP;HTTP/1.1 status=200 response=OK;
Jul 18, 2018 08:52:31.598645000 CET;Host=192.168.1.4;Dest=192.168.1.
44;Protocol=HTTP;HTTP/1.1 status=200 response=OK data=<script lang
uage="javascript"><!-- function dept_onchange(frmselect) {frmselect.
submit();} //--></script><head><title>Wemos1</title></head>;

Example of regular expression:
^(?<date>\w{3} \d{2}, \d{4} \d{2}:\d{2}:\d{2}.\d{3})\d{6}\s(CET);(?
<param1>(\w+=\d+\.\d+\.\d+\.\d+));(?<param2>(\w+=\d+\.\d+\.\d+\.\d+
));(?<param3>[^;]+);(?<param4>[^=]+=[A-Z]{3,4})\s(?<param5>(Uri=)(
?<label>[^?]+))\sHTTP/1.1;$
```

Fig. 2: Example of 4 raw messages collected from a connected thermostat device. The regular expression retrieves a label and 5 parameters here. The label expression will be the label of the action in the formatted trace.

Figure 3 gives, in $Traces(\text{SUL})$, a complete formatted trace, composed of 16 actions, which was extracted by means of 4 regular expressions that assign the called URL and the HTTP responses to labels, and keep some data, e.g., the temperature with the parameter *svalue*. The second step of COnfECt tries to identify distinctive component behaviours in the traces of $Traces(\text{SUL})$. It processes traces and computes a *Correlation coefficient* with the factor $f$, which assesses the degree of correlation of successive actions in a trace. Intuitively, when two successive action sub-sequences are not correlated, then we consider that they come from two distinctive components. We propose several means to define the Correlation coefficient in Section 5. Let us suppose that the trace of Figure 3 has been analysed and that COnfECt has detected the 3 sub-sequences in bold. This means that a first component has produced the first action of the trace. Then, a second component has been invoked. The latter has produced the two actions in bold (/json.htm and Response) and has terminated its execution. The first component has proceeded its execution and so forth. These special sequences in bold are extracted and replaced by the synchronisation actions *call* and *return*, which express that a component has been invoked. The extracted sub-sequences are placed into new trace sets. In our example, 4 trace sets T1-T4 are built. At the end of this step, COnfECt returns a set $STraces$ composed of these trace sets.

In the beginning of the third step (STEP 3A LTS Generation in the figure), every set of STraces is transformed into a LTS by converting traces into LTS paths, which are then joined on the initial state only. Together, these LTSs form an initial system of LTSs. In our example, as we have 4 trace sets, we obtain a system of 4 LTSs. These LTSs include synchronisation actions starting with call and return. Given two LTSs $C_1$ and $C_2$, when the transition $q \xrightarrow{call_C_2} q'$ of the LTS $C_1$ is fired, we say that $C_1$ calls the LTS $C_2$. This action means that the current execution is being paused while another LTS $C_2$ starts its execution at its initial state. When the transition $q \xrightarrow{call_C_2} q'$ of the LTS $C_2$ is fired, we say that $C_2$ is called. The execution of $C_2$ ends once the transition $q \xrightarrow{return_C_2} q'$ in $C_1$ or $q \xrightarrow{return_C_2} q'$ in $C_2$ is fired. Now, COnfECt proposes three strategies for synchronising these LTSs together. The main purpose of these strategies is to address challenge 2,

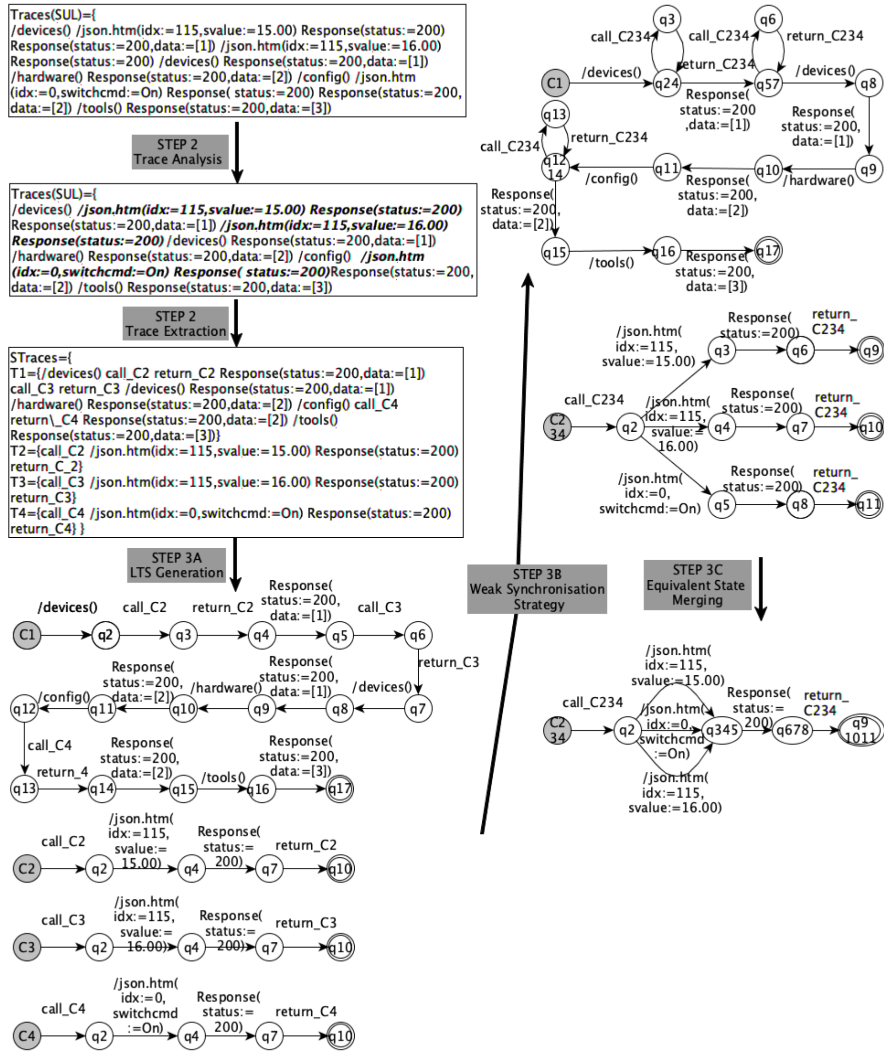



Fig. 3: The COnfECt approach overview

which stems from the fact that the traces collected from SUL usually do not capture all of its valid behaviours. The strategies make possible the generation of models that accept more behaviours by adapting the component integration differently. The first strategy called *Strict synchronisation* only reduces the LTSs by applying the kTail approach to merge equivalent states. The second strategy, called *Weak synchronisation*, tries to detect the components that have similar behaviours by means of a *LTS similarity coefficient*, defined by the factor $f'$. In addition, the transition sequences $q_1 \xrightarrow{call_C_i.return_C_i} q_2$ are replaced by loops to allow repetitive component calls. Figure 3 illustrates the use of this strategy. The LTSs of the components C2-C4 are detected as similar and are joined. This gives the new LTS C234. Then, the LTS C1 is modified to allow repetitive component

calls (STEP 3B). Finally, kTail is applied on the resulting LTSs to merge equivalent states. Several states are merged in this example, e.g., q3, q4, q5 (STEP 3C). These final LTSs capture more behaviours than those given by the Strict strategy. The last strategy, called *Strong Synchronisation*, provides even more general LTSs by returning systems of LTSs such that every LTS allows the invocation of its components at any of its states. In the example of Figure 3, the LTS C1 calls the LTSs C2-C4. Therefore, the Strong strategy adds transition loops of the form $q_1 \xrightarrow{call_C_i.return_C_i} q_1 (2 \leq i \leq 4)$ at every state of C1. The LTSs C2-C4 are unmodified as they do not call other LTSs.

These steps are detailed in the next sections.

## 5 The COnfECt Approach

### 5.1 Step 1: Trace Formatting

COnfECt takes raw messages that are totally ordered by means of their timestamps. These messages are firstly parsed and analysed with regular expressions to retrieve the actions performed by SUL and their related data. We consider that these expressions transform a message into an action of the form $a(\alpha)$ with $a$ a label and $\alpha$ an assignment of some parameters. For example, the action $swith(id := 115, cmd := on)$ is made up of the label "switch" followed by the assignment of two parameters. These regular expressions may also be used to filter out irrelevant messages. A manual analysis of the messages is often required by end-users to derive regular expressions. Although this task may be carried out with little effort on messages collected from small systems, it is known that this may become impractical when SUL is large or complex. Several works addressed this problem (Mariani and Pastore, 2008; Fu et al, 2009; Makanju et al, 2012; Vaarandi and Pihelgas, 2015; Messaoudi et al, 2018; Zhu et al, 2018) and proposed approaches and tools that automatically mine patterns from log files. These patterns may be used to quickly derive regular expressions.

Then, COnfECt proposes four ways to split a list of actions into traces: by requesting a trace identifier, by inspecting timestamps, or by applying the two ordering combinations of these two options. The first mode, proposed by several model learning approaches, combines actions having the same identifier into the same execution trace. The second mode analyses the timestamps of every pair of successive actions and computes means of time intervals. Then, it searches for gaps (distinctive longer durations), which are usually observed when an execution trace ends and another one begins. The detection of these gaps is used for the trace recognition and extraction.

At the end of this steps, we assume having a trace set denoted $Traces(\text{SUL})$, which gathers traces of the form $a_1(\alpha_1)...a_k(\alpha_k)$.

### 5.2 Step 2: Trace Analysis & Extraction

This step identifies component behaviours in the traces of $Traces(\text{SUL})$, it splits them and returns a set $STraces = \{T_1, \ldots, T_n\}$ such that a trace set $T$ of $STraces$ includes traces of one component. Algorithm 1, which implements this

**Algorithm 1:** Trace refinement Algorithm

```
input  : Traces(SUL) = {σ_1, ..., σ_m}
output: STraces = {T_1, ..., T_n}
1 T_1 = {};
2 STraces = {T_1};
3 foreach σ ∈ Traces(SUL) do
4 |  σ'_1 σ'_2 ... σ'_k = Inspect(σ);
5 |  STraces = Extract(σ'_1 σ'_2 ... σ'_k, T_1);
6 return STraces;
```

steps, is mostly based on two procedures. The procedure *Inspect* covers the traces of *Traces*(SUL) and segments them into sub-sequences. These sequences are extracted and placed into new trace sets in *STraces* by the procedure *Extract*. The trace sets of *STraces* will produce LTSs. These procedures are explained below.

### *5.2.1 Trace analysis (procedure Inspect)*

The fundamental idea of COnfECt is that a component should be recognizable by its behaviour in comparison to the behaviours of the other components. We hence cover the traces of SUL with a Correlation coefficient, which helps recognise different component behaviours. This coefficient evaluates the correlation of action sequences in the traces of *Traces*(SUL), i.e. the degree to which successive actions are related according to all the traces of *Traces*(SUL). We want a flexible coefficient, which could be adapted in accordance to the sort of system under learning and to the knowledge we have about this system. We define the Correlation coefficient between two actions by means of a utility function, which involves a weighting process for representing user priorities and preferences. We have chosen the technique *Simple Additive Weighting* (SAW) (Yoon and Hwang, 1995), which allows the interpretation of these preferences with weights:

**Definition 4 (Correlation coefficient)** Let $a_1(\alpha_1)$, $a_2(\alpha_2) \in \mathcal{L}$ and $f_1, \dots\ f_k$ be correlation factors. $Corr(a_1(\alpha_1), a_2(\alpha_2))$ is a utility function, defined as: $0 \leq Corr(a_1(\alpha_1), a_2(\alpha_2)) = \sum_{i=1}^{k} f_i(a_1(\alpha_1), a_2(\alpha_2)).w_i \leq 1$ with $0 \leq f_i(a_1(\alpha_1),$ $a_2(\alpha_2)) \leq 1$, $w_i \in \mathbb{R}_0^+$ and $\sum_{i=1}^{k} w_i = 1$.

The factors can be general or established with regard to a specific context, e.g., network systems, Web applications, etc. We give below two factor examples:

- $f_1(a_1(\alpha_1), a_2(\alpha_1))$= 1 iff $Id(\alpha_1) = Id(\alpha_2)$ with $Id(\alpha)$ the assignment in $\alpha$ of the parameters that identify every component. Otherwise, $f_1(a_1(\alpha_1), a_2(\alpha_2))$= 0. When this factor is used, it is assumed that components are identified with a parameter set and that this set is known and given;
- $f_2(a_1(\alpha_1), a_2(\alpha_2)) = max(\frac{\text{freq}(a_1 a_2)}{\text{freq}(a_1)}, \frac{\text{freq}(a_1 a_2)}{\text{freq}(a_2)})$ with freq($a_1 a_2$) the frequency of having the two labels $a_1$,$a_2$ one after the other in *Traces*(SUL) and freq($a_1$) the frequency of having the label $a_1$. This factor used in text mining computes the frequency of the term $a_1 a_2$ in *Traces*(SUL) over $a_1$ and over $a_2$ to avoid the bias of getting a low factor when $a_1$ is greatly encountered (resp. $a_2$);

The first factor requires some knowledge about SUL, while the second one is more general. Other factors could also be defined. The factor choice or definition

should be addressed by an expert of SUL. If he/she has a good knowledge about it, he/she can choose the most appropriate factor allowing the component detection in a precise manner. In contrast, if no information about SUL is known, we recommend the factor $f_2$. This factor choice may be seen as a disadvantage of the approach. This is discussed in Section 7.5. Other factors might be defined with regard to the action syntax. For instance, string similarities could be used as factors to correlate actions on their common characters. We refer to (Cohen et al, 2003) for the presentation and definition of some of them.

From this Correlation coefficient, we define two relations to express the notion of strong correlation of actions and action sequences. We say that strong-corr($\sigma_1$) holds when $\sigma_1$ has successive actions that strongly correlate. We also define the weak correlation of two action sequences. $\sigma_1$ $weak-corr$ $\sigma_2$ holds when the last event of $\sigma_1$ does not strongly correlate with the first one of $\sigma_2$. In data and text mining, these notions often depend on the considered context, this is why we use a threshold X in the definition given below. This threshold takes a value between 0 and 1, and needs to be appraised by an expert, for instance after some iterative attempts.

**Definition 5 (Strong and Weak Correlations)** Let $a_1(\alpha_1)$, $a_2(\alpha_2) \in \mathcal{L}$, $\sigma_1 = a_1 \dots a_k \in \mathcal{L}^*$. and $X \in [0, 1]$.

1. $a_1(\alpha_1)$ strong-corr $a_2(\alpha_2) \Leftrightarrow_{def} Corr(a_1(\alpha_1), a_2(\alpha_2)) \geq X$.
2. strong-corr($\sigma_1$) iff $\begin{cases} \sigma_1 = a(\alpha) \in \mathcal{L}, \\ \sigma_1 = a_1(\alpha_1) \dots a_k(\alpha_k)(k > 1) \in \mathcal{L}^*, \forall (1 \leq i < k) : \\ a_i(\alpha_i) \text{ strong-corr } a_{i+1}(\alpha_{i+1}) \end{cases}$
3. $\sigma_1$ weak-corr $\sigma_2$ iff $\begin{cases} \sigma_2 = \epsilon, \\ \sigma_2 = a'_1 \dots a'_l \in \mathcal{L}^* \wedge \neg(a_k \text{ strong-corr } a'_1) \end{cases}$

The trace analysis is performed with the procedure *Inspect* given in Algorithm 2, which covers every trace $\sigma$ of $Traces$(SUL) and potentially segments $\sigma$ into (sub-)sequences such that each sequence $\sigma_1$ has a strong correlation and has a weak correlation with the next sequence $\sigma_2$. We consider that these distinctive sequences $\sigma_1$ $\sigma_2$ express the behaviour of two components, a component produces $\sigma_1$ and calls a second component, which produces $\sigma_2$. In Figure 3 (STEP 2 Trace Analysis), 3 distinctive sub-sequences have been detected within the trace by means of the factor $f_2$. We consider that these sequences reflect the behaviours of other components that produce their own actions among the actions of a first component, which invokes them.

*5.2.2 Trace extraction (procedure Extract)*

The procedure takes the traces of $Traces$(SUL) and extracts the sub-sequences detected previously. Intuitively, the procedure splits two successive sequences that have a weak correlation and adds synchronisation actions of the form $call_C_i$ and $return_C_i$ to model component calls, with $C_i$ referring to a future LTS.

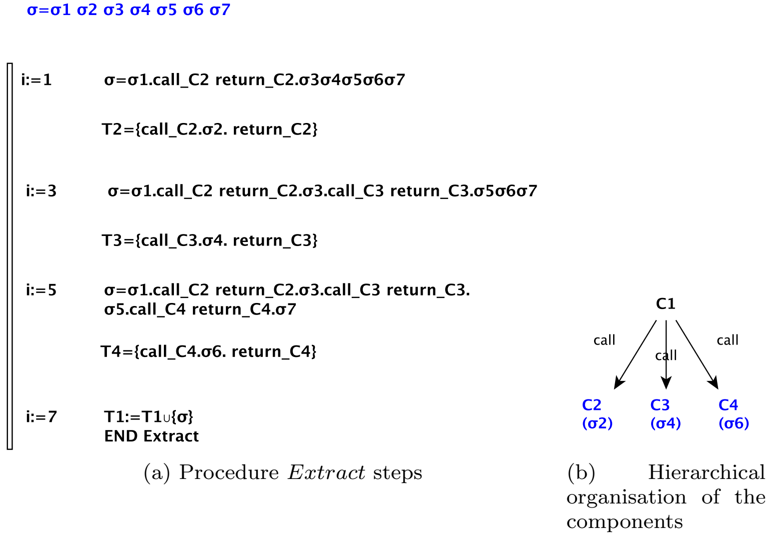
The procedure $Extract(\sigma, T_c, STraces)$ is given in Algorithm 2. It takes a trace $\sigma$, splits it and stores the resulting trace into a set $T_c$. Given a sequence $\sigma_i$ of the trace $\sigma = \sigma_1 \dots \sigma_k$, the procedure $Extract$ tries to find the next sequence $\sigma_j$ such that strong-corr($\sigma_i.\sigma_j$) holds. The sequence $\sigma' = \sigma_{i+1} \dots \sigma_{j-1}$ (or $\sigma' = \sigma_{i+1} \dots \sigma_k$ when $\sigma_j$ is not found) is extracted as it exposes the behaviour of other

**Algorithm 2:** Procedures *Inspect* and *Extract*

```
1  Procedure Inspect(σ) : σ'_1 σ'_2 ... σ'_k is
2      Find the non-empty sequences σ'_1 σ'_2 ... σ'_k such that: σ = σ'_1 σ'_2 ... σ'_k,
         strong-corr(σ'_i)_(1≤i≤k), (σ'_i weak-corr σ'_(i+1))_(1≤i≤k−1);
3  Procedure Extract(σ = σ_1 σ_2 ... σ_k, T_c, STraces): STraces is
4      i := 1;
5      while i < k do
6          n := |STraces| + 1;
7          T_n := {};
8          STraces := STraces ∪ {T_n};
9          σ_p is the prefix of σ up to σ_i;
10         if ∃j > i: strong-corr(σ_i.σ_j) then
11             σ_j is the first sequence in σ_i ... σ_k such that strong-corr(σ_i σ_j);
12             σ := σ_p σ_i.call_C_n return_C_n.σ_j ... σ_k;
13             if (j − i) > 2 then
14                 Extract(σ_(i+1) ... σ_(j−1), T_n);
15             else
16                 T_n := T_n ∪ {call_C_n.σ_(i+1).return_C_n};
17             i := j;
18         else
19             σ := σ_p σ_i.call_C_n return_C_n;
20             if (k − i) > 1 then
21                 Extract(σ_(i+1) ... σ_k, T_n);
22             else
23                 T_n := T_n ∪ {call_C_n.σ_k.return_C_n};
24             i := k;
25     if c ≠ 1 then
26         σ = call_C_c.σ.return_C_c;
27     T_c := T_c ∪ {σ};
28     return STraces;
```

components that are called by the current one. If this sequence $\sigma'$ is composed of only one sub-sequence then it is added to a new trace set $T_n$ of $STraces$. Otherwise, the procedure *Extract* is recursively called with $Extract(\sigma', T_n, STraces)$. In $\sigma$, the sequence $\sigma'$ is removed and replaced by the actions $call_C_n.return_C_n$. After the covering of every sub-sequence of $\sigma$, the procedure *Extract* eventually checks whether $\sigma$ needs to be completed to express that this sequence was produced by a component called by another one: if $T_c$ is not equal to $T_1$ then the trace $\sigma$ is surrounded with $call_C_c$ and $return_C_c$ to express that $\sigma$ stems from a component that was previously called by another one. Otherwise, the sequence $\sigma$ remains unchanged.

Let us illustrate the functioning of the procedure *Extract* with the example of Figure 4a, which takes back the trace of Figure 3. This trace was segmented into seven sequences, which are weakly correlated. We start at $\sigma_1 = /devices()$ (i:=1). The next sequence $\sigma_2 = /json.htm.Response$ expresses the behaviours of another component. But the algorithm has to detect when the component invocation ends. To do so, it looks for the next sequence in the trace that is strongly correlated with $\sigma_1$. This sequence represents the resuming of the first component execution after the invocation of another component. In our example, this next sequence is $\sigma_3 = Response$, which represents the receipt of a response after the action /devices(). The sequence $\sigma_2$ is extracted and replaced by the actions $call_C2$ $return_C2$. The procedure is not recursively called as $\sigma_2$ is not composed of several




(a) Procedure *Extract* steps

(b) Hierarchical organisation of the components

Fig. 4: Sequence extraction example

weakly correlated action sequences. The sequence $\sigma_2$ is now surrounded with the actions $call_C2$ and $return_C2$ to prepare the LTS synchronisation. The resulting sequence is added to the new trace set $T_2$. We go back to the trace $\sigma$ at the sub-sequence $\sigma_3$ (i:=3). The same process is applied on $\sigma_4$ and later on $\sigma_6$ until the algorithm reaches the end of the sequence $\sigma$ (with i:=7). The trace $\sigma$ becomes $\sigma_1.call_C2\ return_C2.\sigma_3.call_C_3\ return_C3.\sigma_5.call_C_4\ return_C4.\sigma_7$. The trace $\sigma$ comes from $Traces$(SUL), which means that $\sigma$ captures the behaviour of the root component (assumption H3) that has not been called by another component. Hence, at the end of the procedure, this trace is not surrounded by synchronisation actions. $\sigma$ is placed into the trace set $T_1$. At the end of the procedure, we have recovered the hierarchical structure of components depicted in Figure 4b. And we get four trace sets, gathered into the set STraces given in Figure 3.

Once the procedure *Extract* terminates, Algorithm 1 yields the set $Straces = \{T_1, T_2, \ldots, T_n\}$ with $T_2, \ldots, T_n$ some sets including one action sequence and $T_1$ a set of modified traces originating from $Traces$(SUL).

### 5.3 Step 3: LTS Synchronisation

This step lifts the traces of $STraces$ to the level of LTSs and proposes three LTS synchronisation strategies, which provide systems of LTSs having different levels of generalisation.

Given the trace set $T \in STraces$, a trace $\sigma = a_1 \ldots a_k$ of $T$ is transformed into the LTS path $q0 \xrightarrow{a_1 \ldots a_k} q_k$ such that the states $q_1 \ldots q_k$ are new states. These paths are joined by a disjoint union on the state $q0$ to build a LTS having a tree form:

**Definition 6 (LTS generation)** Let $T = \{\sigma_1, \ldots, \sigma_m\}$ be a trace set. $C = \langle Q, q0, \Sigma, \rightarrow\rangle$ is the LTS derived from $T$ where:

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

$$\frac{\sigma_{id}{=}a_1(\alpha_1)...a_k(\alpha_k)}{q0 \xrightarrow{a_1(\alpha_1)} q_{id1}...q_{idk-1} \xrightarrow{a_k(\alpha_k)} q_{idk}}$$

Once every trace set of $STraces$ is transformed into a LTS, we have a first system of LTSs $SC = \langle S, C\rangle$ with $C$ the set of LTSs derived from $STraces$ and $S$ the set of synchronisation actions of the form $call_C_i$ and $return_C_i$, found in the action sets of the LTSs.

The previous step of COnfECt has segmented and extracted the traces of $Traces$(SUL) in such a way that they include synchronisation actions. These actions were added to prepare the synchronisation of components with LTSs.

We now propose three strategies, which adapt the transitions labelled by synchronised actions to answer to Challenge 2. These are implemented in Algorithm 3.

**Algorithm 3:** LTS synchronisation strategies

```
   input : System of LTSs SC = ⟨S, C⟩ with C = {C_1, ..., C_n}, strategy
   output: System of LTSs SC_f = ⟨S_f, C_f⟩
 1 if strategy = Strict Synchronisation then
 2     return kTail(k = 2, SC);
 3 else
 4     ∀(C_1, C_2) ∈ C^2 Compute Similarity_LTS(C_1, C_2);
 5     Build a similarity matrix;
 6     Group the LTSs into clusters {Cl_1, ... Cl_k} such that ∀(C_1, C_2) ∈ Cl_i^2 : C_1 similar C2 ;
 7     foreach cluster Cl = {C_1, ..., C_l} do
 8         C_Cl:=Disjoint Union of the LTSs C_1, ..., C_l;
 9         C_f = C_f ∪ {C_Cl};
10     foreach C_i = ⟨Q, q0, Σ, →⟩ ∈ SC_f do
11         foreach q_1 -a-> q_2 with a = call_C_m or a = return_C_m do
12             Find the Cluster Cl such that C_m ∈ Cl;
13             Replace C_m by C_Cl in the label a;
14             S_f := S_f ∪ {a};
15         foreach q_1 -call_C_m return_C_m-> q_2 ∈→ do
16             Merge (q_1, q_2);
17     if strategy = Strong Synchronisation then
18         foreach C_i = ⟨Q, q0, Σ, →⟩ ∈ C_f do
19             Complete the outgoing transitions of the states of Q so that C_i is callable-complete;
20             foreach q_1 -call_C_m return_C_m-> q_2 ∈→ do
21                 Merge (q_1, q_2);
22     return kTail(k = 2, SC_f)
```

## 5.3.1 Strict Synchronisation

Algorithm 1 has previously segmented every trace of $Traces$(SUL) into sub-sequences of actions. When a sub-sequence is extracted, it is placed into a new trace set in $STraces$ and replaced by the actions $call_C_i.return_C_i$. The LTSs of $SC$, derived from $STraces$, do not repetitively call other LTSs and are composed of

acyclic paths only. We call this LTS configuration, Strict synchronisation. This strategy, which is mostly and implicitly implemented in Algorithm 1, eventually calls the kTail algorithm to merge the similar states found in the LTSs of $SC$. This strategy limits over-generalisation, i.e. the fact of generating models expressing more behaviours than those given in the initial trace set $Traces$(SUL). This is more formally captured by the following proposition, which postulates that, before calling kTail, the traces of $SC$ leading to final states are the traces of $Traces$(SUL).

**Proposition 1** *Let $SC = \langle S, C\rangle$ be a system of LTSs achieved with the Strict synchronisation strategy (before the call of kTail), with $C = \{C_1, \dots, C_n\}$. $QF$ is the set of final states of the LTS $C_1 \parallel C_2 \parallel \cdots \parallel C_n$.*
*$Traces_{QF}(SC) = Traces(\mathrm{SUL})$.*

*5.3.2 Weak Synchronisation*

This strategy aims at reducing the number of LTSs and allows repetitive component calls. Algorithm 1 may indeed have refined too much $Traces$(SUL), hence the system of LTSs $SC$ might include several LTSs modelling the functioning of the same component. This strategy attempts to gather these LTSs by means of a LTS Similarity coefficient, which evaluates the similarity of two LTSs. Like the Correlation coefficient, the LTS similarity is defined with a utility function and factors to be compatible with different sorts of systems:

**Definition 7 (LTS Similarity Coefficient)** Let $C_i = \langle Q_i, q0_i, \Sigma_i, \rightarrow_i\rangle$ $(i = 1, 2)$ be two LTSs of the system of LTSs $SC = \langle S, C\rangle$. Let also $f'_1, \dots\ f'_k$ be LTS similarity factors. The LTS Similarity of $C_1$, $C_2$ is defined as:
$0 \leq Similarity_{LTS}(C_1, C_2) = \sum_{i=1}^{k} f'_i(C_1, C_2).w_i \leq 1$ with $0 \leq f'_i(C_1, C_2) \leq 1$, $w_i \in \mathbb{R}_0^+$ and $\sum_{i=1}^{k} w_i = 1$.
$C_1$ similar $C_2 \Leftrightarrow_{def} Similarity_{LTS}(C_1, C_2) \geq Y$, with $Y \in [0, 1]$.

We provide two similarity factors below. The first one refers once again to the component identification, just like the correlation factor $f_1$. The second factor measures the similarity of two LTSs with regard to the actions they share.

- $f'_1(C_1, C_2)$= 1 iff $\forall a_1(\alpha_1), a_2(\alpha_2) \in (\Sigma_{C_1} \cup \Sigma_{C_2}) \setminus S$, $Id(\alpha_1) = Id(\alpha_2)$, with $Id(\alpha)$ the assignment in $\alpha$ of the parameters that identify every component. Otherwise, $f'_1(C_1,\ C_2) = 0$. This implies that two similar LTSs must have actions including the same component identification. The factor is not applied on the synchronised actions of $S$, which were added by the previous step of COnfECt;
- $f'_2(C_1, C_2) = Overlap(\Sigma_{C_1} \setminus S, \Sigma_{C_2} \setminus S)$, with the overlap of two sets $A$ and $B$ defined by $|A \cap B|/min(|A|, |B|)$. Several general similarity coefficients are available in the literature for comparing the similarity and diversity of sets, e.g., the coefficients Jaccard or SMC (Tan et al, 2005). We have chosen the Overlap coefficient because the action sets of two LTSs may have different sizes.

The Weak synchronisation strategy is implemented in Algorithm 3 lines (3-16). It computes the LTS Similarity of every pair of LTSs of $SC$. The similar LTSs are then grouped by means of a clustering technique, which uses the LTS Similarity coefficients. The LTSs of the same cluster are joined with a disjoint

union. Furthermore, the labels of the transitions $q_1 \xrightarrow{call_C} q_2$, $q'_1 \xrightarrow{return_C} q'_2$ are updated accordingly so that the correct LTSs are being called (Algorithm 3 lines(11-14)). In addition, every sequence $q_1 \xrightarrow{call_C\ return_C} q_2$ is replaced by a loop $(q_1, q_2) \xrightarrow{call_C\ return_C} (q_1, q_2)$ by merging both states $q_1$ and $q_2$.

*5.3.3 Strong Synchronisation:*

This strategy aims at providing more general models than the Weak strategy, by assuming that a component $C_1$, which requests services from other components, may repetitively call them at any of its states. We denote $R$ the set of LTSs modelling components that are invoked by $C_1$. We define that $C_1$ is callable-complete when $C_1$ may call any LTS $C_2$ of $R$ at any of its states:

**Definition 8 (Callable-complete LTS)** Let $SC = \langle S, C \rangle$ be a system of LTSs and $C_1 = \langle Q_1, q0_1, \Sigma_1, \rightarrow_1 \rangle \in C$. $R$ stands for the set of LTSs sharing synchronised actions with $C_1$. $R = \{C_i \in C \mid q_1 \xrightarrow{call_C_i} q_2 \in \rightarrow_1\}$.

$C_1$ is callable-complete iff $\forall q \in Q_1, \forall C_2 \in R, \exists q' \in Q_1 : q \xrightarrow{call_C_2.return_C_2} q'$.

The strategy is implemented in Algorithm 3 lines(3-21). As with the Weak Synchronisation strategy, the similar LTSs of $SC$ are assembled into bigger LTSs and the transitions labelled by synchronisation actions are updated accordingly. Additionally, every state $q$ of the LTSs is completed with new outgoing transitions of the form $q \xrightarrow{call_C\ return_C} q$ so that the LTSs of $SC$ become callable-complete.

The Weak and Strong synchronisation strategies produce more general systems of LTSs than the first strategy. This is captured by this proposition:

**Proposition 2** *Let $SC = \langle S, C \rangle$ be a system of LTSs achieved with the Weak or Strong synchronisation strategy (before the call of kTail), with $C = \{C_1, \dots, C_n\}$. $QF$ is the set of final states of $C_1 \parallel C_2 \parallel \cdots \parallel C_n$.*
$Traces_{QF}(SC) \supset Traces(\mathrm{SUL})$.

For the three strategies, the LTSs of $SC = \langle S, C \rangle$ may include equivalent states, which should be joined to generate more concise models. As stated previously, we use the kTail approach, which merges the states that share the same k-future. We use $k = 2$ as recommended by Lorenzoli et al (2008); Lo et al (2012).

Figure 3 illustrates the use of the Weak strategy. Each trace set of $STraces$ is firstly transformed into a LTS (STEP 3A). As the trace sets are composed of only one action sequence, we get LTSs having one path. Then, the similar LTSs have to be grouped. To define the LTS Similarity coefficient, we choose the factor $f'_2$. We compute a similarity matrix by means of the LTS Similarity coefficient. Figure 5 shows the matrix obtained with the four LTSs of our example. If we set the LTS similarity threshold $Y$ to 0,5, we observe that two classes of similar LTSs emerge in this matrix: $(C_1)$ and $(C_2, C_3, C_4)$. A clustering technique, e.g., the Ward's method (Willett, 1988), can help automate this grouping of similar LTSs. The similar LTSs are then joined by means of a disjoint union. As we choose the Weak synchronisation strategy, the transition sequences of the form $q_1 \xrightarrow{call_C_m return_C_m} q_2$ have been replaced with loops in C1. We finally obtain

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|---|
| $C_1$ | 1 | 0 | 0 | 0 |
| $C_2$ | 0 | 1 | 0.5 | 0,5 |
| $C_3$ | 0 | 0.5 | 1 | 0,5 |
| $C_4$ | 0 | 0,5 | 0,5 | 1 |

Fig. 5: LTS Similarity matrix example

two LTSs (STEP 3B): $C_1$ expresses the use of the Web interface, $C_{234}$ models the component that sends data (temperature, motion detection) to a server. The LTS $C_{234}$ holds three equivalent state classes $(q3, q4, q5)$, $(q6, q7, q8)$ and $(q9, q10, q11)$, which are merged with kTail (STEP 3C).

## 6 Implementation

Our approach is implemented in Java and is released as open source. The prototype tool consists of two applications. The first step of COnfECt, which performs the trace formatting by means of regular expressions, is implemented in a first tool called *TFormat*[1]. But, end-users might prefer using their own trace formatting tool like LogParser, which automatically learns event templates from unstructured logs (Zhu et al, 2018).

The second application[2] performs the last steps of the approach. The user gives as inputs a folder containing formatted trace files, the chosen factors, the related thresholds and a LTS synchronisation strategy. For the Weak and Strong strategies, we use a clustering approach based on the Ward's method, which is a well-known agglomerative hierarchical clustering method. In short, the LTS clustering is carried out as follows: 1) each LTS is placed into its own initial cluster and similarity coefficients are computed; 2) the two clusters that have the closest similarity (greater than the given threshold $Y$) are merged, similarity coefficients are updated and so forth until there is no more similar cluster. This approach avoids the generation of too large clusters and does not need to pre-specify the cluster number.

## 7 Preliminary Evaluation

With this implementation of COnfECt, we conducted several experiments in order to evaluate the following criteria:

- C1 (Component detection): is COnfECt able to detect the correct number of components? The key contribution of COnfECt is its ability of detecting sub-sequences in traces and to link them to separate components. We studied C1 with a real device that we implemented and whose internal architecture is known. Our knowledge about the system under learning allowed us to study the inferred LTSs and to check whether these do not capture mixing behaviours of several real components;

[1] https://github.com/sasa27/TFormat
[2] https://github.com/Elblot/COnfECt-2.0

- C2 (Relevance of the models): is COnfECt able to infer concise and readable models, which express system behaviours and reject abnormal behaviours? C2 investigates how the inferred models accept valid traces including new traces not used for the model generation and the capability of these models to reject invalid traces. We compare COnfECt to CSight and kTail, as kTail is used as baseline in several papers, e.g., (Lo et al, 2012; Ohmann et al, 2014);
- C3 (Efficiency/Scalability): how long does COnfECt take to generate systems of LTSs? How does COnfECt scale with the size of the trace set? We study the efficiency and scalability of COnfECt, compared to CSight and kTail.

## 7.1 Empirical Setup

For this evaluation, we chose a real system that we implemented to be able to appraise the accuracy of the generated models. The system under learning is the connected thermostat taken as example in the paper, whose source code has been made available[3]. This IoT device controls heating pumps according to external events and integrates 3 components: a sensor manager coordinating 4 physical sensors, a component that updates the internal clock of the device by calling a NTP server, and a Web server allowing the configuration of the device and the reading of data, e.g. the temperature. These components meet the requirements given in Section 4 and can be monitored to collect HTTP traces. This device has been implemented in such a way that each component may be turned on or off without blocking the functioning of the others. This feature is important for this evaluation to derive several different models as the the effect that events have on the behaviour of the system depend on the set of components being activated. For instance, if the physical sensors are turned off, then the thermostat will not start heating pumps when the temperature is below a given threshold. We ran this IoT device with several component configurations using 1 to 3 components. The HTTP traces were formatted with our tool TFormat and 10 regular expressions. These traces have the same form as the trace given in Figure 2. The trace sets are available here[4]. The LTS generation was performed on a desktop computer with 1 Intel(R) CPU i5-6500 @ 3.2GHz and 16GB RAM.

We adapted the smallest trace set collected previously to compare COnfECt and CSight. But we were unable to have results with CSight after 5 hours of computation, which was our limit for each experiment. We observed that the first steps of CSight were achieved, but the model-checker returned successive time-outs while the model refinement step. We suspect that the model-checker is unable to check the invariant satisfiability on large trace sets. Therefore, to compare CSight and COnfECt, we have taken back two trace sets available with the CSight implementation. The first one was extracted from TCP logs, and the second one from logs of the AlternatingBit protocol.

### *7.1.1 Factor choice & thresholds assessment*

The Correlation and LTS Similarity coefficients have to be defined by setting factors, weights and thresholds. We took for the experiments the factor combinations

[3] https://github.com/sasa27/OpenThermostat
[4] https://github.com/Elblot/COnfECt-2.0

| Configuration | # real Components | Factors | Strict | Weak | Strong |
|---|---|---|---|---|---|
| Conf. 1 | 1 | $f_2 \geq 1; f_2' \geq 0.75$ | 1 | 1 | 1 |
| Conf. 2 | 1 | $f_2 \geq 1; f_2' \geq 0.75$ | 1 | 1 | 1 |
| Conf. 3 | 1 | $f_2 \geq 1; f_2' \geq 0.75$ | 1 | 1 | 1 |
| Conf. 4 | 2 | $f_2 \geq 0.5; f_2' \geq 0.75$ | 52 | 2 | 2 |
| Conf. 5 | 2 | $f_2 \geq 0.6; f_2' \geq 0.75$ | 82 | 2 | 2 |
| Conf. 6 | 2 | $f_2 \geq 0.6; f_2' \geq 0.75$ | 74 | 2 | 2 |
| Conf. 7 | 3 | $f_2 \geq 0.75; f_2' \geq 1$ | 342 | 342 | 342 |
| Conf. 8 | 3 | $f_2 \geq 0.5; f_2' \geq 1$ | 143 | 143 | 143 |
| Conf. 9 | 3 | $f_2 \geq 0.5; f_2' \geq 0.75$ | 143 | 3 | 3 |
| Conf. 10 | 3 | $f_1 \geq 1; f_1' \geq 1$ | 104 | 3 | 3 |
| TCP1 | 2 | $f_1 \geq 1; f_1' \geq 1$ | 29 | 2 | 2 |
| Alt.Bit1 | 2 | $f_1 \geq 1; f_1' \geq 1$ | 100 | 2 | 2 |
| TCP2 | 2 | $f_2 \geq 1; f_2' \geq 0.49$ | 38 | 1 | 1 |
| Alt.Bit2 | 2 | $f_2 \geq 1; f_2' \geq 0.95$ | 88 | 2 | 2 |

Table 1: Number of components detected by COnfECt.

$f_1/f_1'$ and $f_2/f_2'$. The factors $f_1/f_1'$ require that an expert of the system provides the parameters allowing the identification of all the components of SUL. The single thresholds we used with $f_1/f_1'$ are $X = Y = 1$, which intuitively means that two action sequences are strongly correlated or that two LTSs are similar iff they share the same component identification. We applied this factor combination on the IoT device and on the logs of the TCP and AlternatingBit protocols and obtained 3 configurations (Conf. 10, TCP1, Alt. Bit1) given in Table 1. The factor combination $f_2/f_2'$, which is based on the labels found in traces, does not require to have any specific information about the system under learning. But the choice of the thresholds has a strong influence on the accuracy of the models. An expert of the system should assess this accuracy and the thresholds. For the experiments, we applied this protocol:

1. generation of the first models with the default thresholds $X \geq 0.75, Y \geq 1$;
2. analysis of the models generated with the Strict strategy. If $|Straces|$ is lower than the expected number of components or if we observe in the traces of $Straces$ some action sequences that seem to belong to several components, then increase the threshold $X$. Conversely, decrease $X$. To find the most appropriate value, take $X = 1$ or $X = 0.1$ and follow a bisection method;
3. when the Weak or Strong synchronisation strategy is chosen, analysis of the generated LTSs. If two LTSs seem to capture the behaviours of the same component, then decrease $Y$. To find the most appropriate value, take $Y = 0.1$ and follow a bisection method.

We applied this protocol on the IoT device and the two protocols with the configurations Conf. 1 to 9, TCP2 and Alt.Bit2. given in Table 1. Conf. 7 to 9 show the three steps of the protocol.

Finally, we collected and formatted a set of 20 traces (resp. 50) composed of about 50 actions with Conf. 1-6 (resp. Conf. 9,10) and used a set of 10 traces (98 actions) with the AlternatingBit and TCP case studies.

### 7.2 C1 (Component detection)

Table 1 lists the number of LTSs inferred by COnfECt. For comparison purposes, we also recall the exact number of components for each system configuration.

The lines Conf. 1-6, 9,10 show the results achieved with COnfECt when the thresholds $X$ and $Y$ are correctly set after following the protocol defined in Section 7.1.1. The approach detects a correct number of components whatever the strategy used in Conf. 1 to 3. With Conf. 4-6, 9 and 10, the Strict strategy provides too many LTSs because of the second step of COnfECt, which refines the traces too much. But, the Weak and Strong strategies provide a correct component number because they assemble the similar LTSs together.

Conf. 7 to 9 illustrate the incremental use of COnfECt to detect the appropriate thresholds $X$ and $Y$. The component detection is false whatever the strategy used in Conf. 7 and 8. In Conf. 7, we observed that the initial traces were too much segmented. We hence decreased the threshold $X$ to 0.6 for the Correlation coefficient and reran COnfECt. With Conf. 8, we detected that no similar LTSs were detected and decreased the threshold $Y$ to 0.75. With Conf. 9, COnfECt detects the correct number of components with the two last strategies. Thanks to our knowledge about the SUL implementation, we manually analysed the LTSs built with the configurations and strategies giving a correct number of components. We did not detected any mixture of component behaviours, and observed that each LTS expresses the behaviours of a real component.

With the configurations TCP2 and Alt.Bit2, COnfECt cannot detect component behaviours. As the factor $f_2$ computes term frequencies, its accuracy depends on the trace set size. With these configurations, we observed that the trace sets are too small for calculating relevant frequencies. Either the number of detected components is false (TCP2) or the models are incorrect (Alt.Bit2).

With Conf. 10, TCP1 and Alt.Bit1, the trace segmentation and the LTS similarity are based on the component identification (factors $f_1/f'_1$). With these configurations, the number of components is correctly detected with the Weak and Strong strategies, without adjusting any threshold like with $f_2/f'_2$.

These experiments show that COnfECt answers to Challenge 1, but for the factor combination $f_2/f'_2$, it is required to have a large trace set and to adjust the factor thresholds. The general functioning of COnfECt is illustrated in Conf. 4-6, 9 and 10: the Strict strategy refines the traces and often returns too many LTSs. The two last strategies counterbalance the trace refinement.

### 7.3 C2 (Relevance of the models)

Regarding the results of Table 1, it is worth noting that we infer irrelevant models if the given thresholds do not allow a correct component detection. As stated earlier, the threshold choice difficulty depends on the factors. For instance, the factors $f_1/f'_1$ can each take two values (0 or 1). In contrast, $f_2/f'_2$ have to be evaluated with several model generation attempts.

#### *Valid and invalid trace acceptance*

We firstly analysed the ability of the generated models in accepting valid and invalid traces. The former are traces collected from the system under learning, which were not used for the model generation. The latter are traces including unexpected actions that should be rejected by models.

As the model quality depends on the approaches, the coefficient thresholds and strategies, we chose to check the trace acceptance on the 30 models produced by kTail, COnfECt and CSight with the configurations Conf. 4-6, 9,10, TCP1 and Alt.Bit1. For COnfECt, we recall that the models generated with these configurations capture the behaviours of two or more components and are built with correct thresholds. 70% of the traces collected in these configurations were used for inferring models, the rest used as valid traces for testing the models.

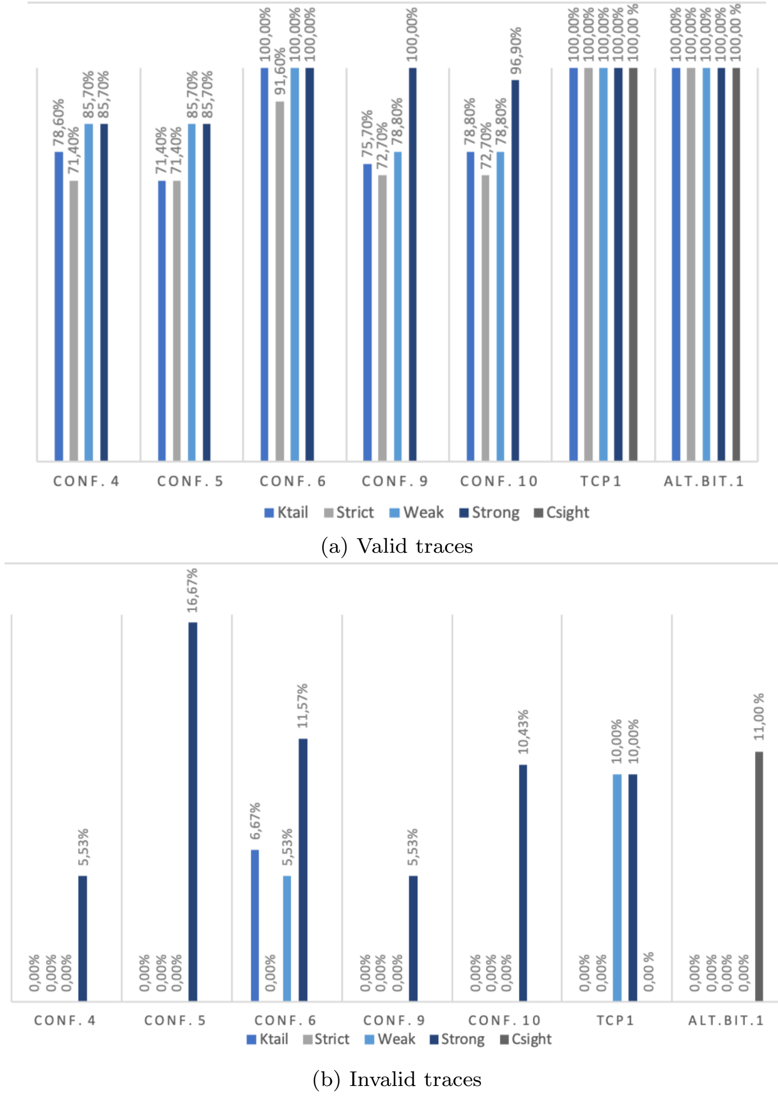
Then, we automatically generated invalid traces by applying three mutations on the valid ones: random repetitions of actions, inversion of HTTP requests and responses, and modifications of the strongly correlated sequences in traces. This last modification was performed on traces after the second step of COnfECt: for two consecutive sequences $\sigma_1 = a_1 \dots a_k$ $\sigma_2 = a'_1 \dots a'_k$, we inverted the two actions $a_k$ and $a'_1$. We produced 120 traces of 40 actions for Conf. 4-6, 9-10 and 30 traces of 20 actions for the configurations TCP1 and Alt.Bit1.

Figure 6a illustrates the rates of valid traces accepted by the models. In comparison to kTail, the Strict strategy of COnfECt gives models that accept slightly less valid traces. The systems of LTSs obtained with this strategy are indeed less general because of the partitioning of the initial trace set (less states are merged in final models). Whatever the configuration, the models inferred by the two last strategies of COnfECt accept more valid traces than the models of kTail. This increase is a consequence of allowing repetitive component calls in the LTSs. Unsurprisingly, the Strong strategy provides the systems of LTSs that accept the highest rates of valid traces (between 90 and 100 %) because these LTSs are callable-complete. For the two last configurations, we observe that 100% of the traces are accepted by the models, whatever the approach used. We suspect here that the initial trace sets are not sufficiently large to observe differences between the approaches. We tried to increase these traces sets, but CSight has not been able to return a model. Figure 6b depicts the rates of invalid traces accepted by the models given by kTail, COnfECt and CSight. We observe that the Weak strategy and kTail provide the same rates with Conf 1. to 10. The Strong strategy builds models that accept between 5,5 and 17 % of invalid traces. After inspection, these invalid traces are mostly composed of repetitive HTTP requests, which are accepted because the models are callable-complete. CSight provides more correct models than COnfECt with TCP1, but less correct models with Alt.Bit1.

From these experiments, we conclude that COnfECt, with the Weak and Strong strategies, outperforms kTail and is at least as efficient as CSight. The Weak strategy provides models that accept slightly more valid traces than kTail and rejects the same amount of invalid traces as the other approaches. The Strong strategy tends to give models that accept more valid traces but also more invalid ones.

*Model readability*

We evaluated the readability of the models generated by COnfECt, CSight and kTail by measuring the model sizes. The first six columns of Table 2 give the number of states and transitions with these configurations. As expected, we obtain bigger LTSs with COnfECt than the ones inferred with kTail and CSight (except with Conf. 1-3 since SUL has only one component). With the two last configurations, we observe that the models inferred by CSight and kTail are close in size




(a) Valid traces



(b) Invalid traces

Fig. 6: Rates of traces accepted by models.

and much more concise than the models of COnfECt. This outcome stems from our algorithm, which completes LTSs with transitions labelled by synchronisation actions. With the Strict strategy, the state number is increased by 1520 % because many LTSs are built, are not joined later, and few equivalent states are found in these LTSs. We observed here that this strategy returns too much LTSs with large trace sets and should be restricted to small trace sets only. The state number is increased by 50 % with the Weak strategy and remains the same with the Strong one.

The transition labelled by synchronisation actions help interpret the component combination and are required to later compose LTSs. But, these are not

| Conf. | kTail | | CSight | | Strict | | Weak | | Strong | | Strict+hide | | Weak+hide | | Strong+hide | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #st | #tr | #st | #tr | #st | #tr | #st | #tr | #st | #tr | #st | #tr | #st | #tr | #st | #tr |
| Conf. 1 | 31 | 64 | / | / | 31 | 64 | 31 | 64 | 31 | 64 | 31 | 64 | 31 | 64 | 31 | 64 |
| Conf. 2 | 7 | 12 | / | / | 7 | 12 | 7 | 12 | 7 | 12 | 7 | 12 | 7 | 12 | 7 | 12 |
| Conf. 3 | 4 | 4 | / | / | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Conf. 4 | 53 | 114 | / | / | 620 | 598 | 77 | 158 | 61 | 144 | 402 | 354 | 63 | 113 | 36 | 64 |
| Conf. 5 | 59 | 117 | / | / | 740 | 699 | 83 | 161 | 62 | 141 | 435 | 371 | 67 | 113 | 37 | 58 |
| Conf. 6 | 24 | 54 | / | / | 438 | 422 | 34 | 79 | 23 | 50 | 249 | 214 | 24 | 52 | 7 | 11 |
| Conf. 9 | 92 | 229 | / | / | 2066 | 1916 | 155 | 360 | 75 | 179 | 1195 | 964 | 116 | 235 | 41 | 77 |
| Conf. 10 | 95 | 235 | / | / | 1610 | 1581 | 160 | 382 | 77 | 189 | 1056 | 953 | 117 | 247 | 43 | 81 |
| TCP1 | 17 | 19 | 12 | 19 | 864 | 832 | 23 | 32 | 27 | 44 | 577 | 476 | 16 | 18 | 15 | 17 |
| Alt.Bit1 | 22 | 28 | 28 | 29 | 203 | 174 | 33 | 54 | 36 | 67 | 127 | 86 | 18 | 29 | 18 | 30 |

Table 2: Sizes of the LTSs obtained with kTail CSight and the three strategies of COnfECt. "hide" refers to the removal of the LTS transitions labelled by synchronisation actions. #st and #tr stand for the number of states and transitions.

significant if one wants to focus on the component behaviours only. Table 2 provides, in the last six columns, the number of states and transitions after applying the *hide* operation to remove the transitions labelled by synchronisation actions. The models generated by COnfECt become more concise than those obtained with kTail. More precisely, the state number is increased by 14 % with the Weak+hide strategy in comparison to kTail. But the former divides the system behaviours into several smaller LTSs, which are much more readable. The state number is reduced by 40 % when using the Strong+hide strategy. For instance, the number of states is equal to 41 in Conf. 9, whereas the LTS achieved with kTail has 92 states. The Strong+hide strategy builds models whose sizes are close to the sizes of the models inferred by CSight.

We compared the models generated by COnfECt and CSight with the two last configurations to evaluate their differences. The systems of LTSs of COnfECt are usually less readable as they contain additional synchronisation actions. If we apply the Strong+hide strategy, both CSight and COnfECt generate models of similar sizes though. The other differences of behaviours mainly come from the functioning of the two approaches that do not target the same kind of systems. For example, the figures 7a, 7b illustrate the models *pid*0 *pid*1 inferred by CSight for the AlterbatingBit protocol, and the figures 7c, 7d show the models $C_1$ $C_2$ given by COnfECt. For the first component, the CFSM of CSight is here more concise, but it accepts more invalid behaviours as it allows the successive sending of the same bit instead of incrementing it. For the second component, the models *pid*1 and $C_2$ have the same size but seem different. The initial state of the model *pid*1 of CSight only accepts the input m0, whereas $C_2$ accepts both the actions m0 and m1. This difference comes from these two observations: 1) COnfECt builds more general models with the Strong strategy; 2) COnfECt builds models of components that control other components whereas CSight builds models of autonomous components. Here, the component $C_1$ requests a service to $C_2$ by sends a first action m0, therefore $C_2$ will always start by executing the action m0.

These experiments show that the models inferred by our approach are relevant on the condition that the correct coefficient thresholds are given. The three strategies help manage the generalisation level, which relates to Challenge 2. We showed that the Strong+hide strategy tends to provide readable models that accept the highest ratio of valid traces, with a reasonable ratio of invalid ones. If the user wishes to minimise the over-generalisation problem in models but still wants readable ones, he/she can apply the Weak+hide strategy instead.

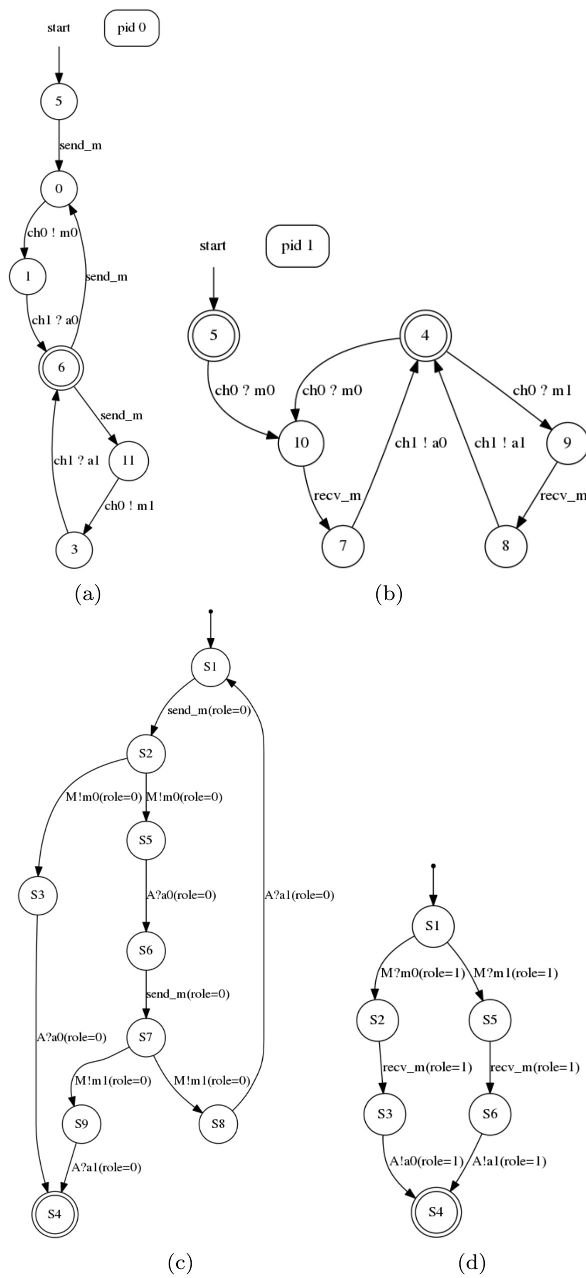
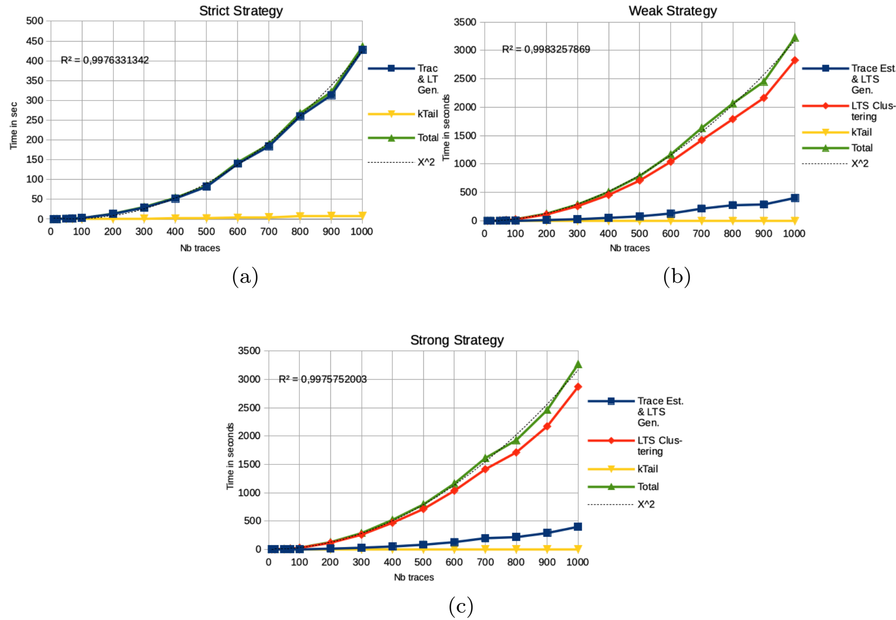



Fig. 7: Models generated by CSight and COnfECt (Strong+hide strategy) for the AlternatingBit protocol

## 7.4 C3 (Efficiency/Scalability)




(a) (b) (c)

Fig. 8: Execution times vs. nb of traces

We experimented COnfECt and kTail with the parameters of Conf. 9 and several trace sets containing from 10 to 1000 traces composed of about 50 actions. As stated earlier, we were unable ro run CSight with these traces. Therefore, we also measured the execution times of Csight, kTail and COnfECt with the two traces sets of the TCP and AlternatingBit protocols.

Our implementation of kTail required less than 1 second to generate models. The execution times of COnfECt are illustrated in Figures 8a-8c and given in seconds. In the figures, the curves "Total" represent the complete execution times. These are detailed with the other curves, which depict the execution times of some sub-steps of COnfECt: Trace Analysis & Extraction, LTS clustering, and call of kTail. With the Strict strategy and trace sets having no more than 100 traces (10, 20, 50, 100), COnfECt builds systems of LTSs in less than 3 seconds. We observed that the evaluation of the factor $f_2$ takes most of the time as the action set needs to be scanned with two nested loops. Hence, its is not surprising to observe that the tendency curve confirms that the time complexity is quadratic. The time executions substantially increase with the Weak and Strong strategies. With 100 traces, the execution time go up to 28 seconds. As the curves "LTS clustering" are close to the curves "Total", we can conclude that the additional time is consumed by the Ward clustering technique, which also has a quadratic complexity. With the traces of the TCP and AlternatingBit protocols, we observed that CSight is significantly slower than COnfECt. The former respectively takes 3552ms and 14657ms to build models, whereas COnfECt takes 48ms and 46ms.

Furthermore, the time-outs we observed on the current CSignt implementation with large trace sets also suggests that CSignt might hardly scale to large systems producing large log files. On the contrary, COnfECt is able to take large trace sets even when we run it on a moderate budget computer. With 50000 actions (1000 traces), the model generation requires around 50 minutes, which remains a reasonable execution time.

Concerning the memory consumption, these experiments required less than 16 Go of memory. If the trace set exceeds 70000 actions, more memory is required. We observed that the space complexity remains linear w.r.t. the trace number.

These results suggest that COnfECt can handle large trace sets and infer models in reasonable time. As the execution time of COnfECt follows a quadratic curve, it is however difficult to claim that it scales well. But the current implementation of COnfECt is absolutely not optimised: the algorithm Trace Analysis & Extraction could be parallelised. The Ward clustering technique could also be replaced by another algorithm having a lesser complexity.

### 7.5 Threats to Validity

There are many application and system contexts, but this preliminary experimental evaluation is only applied on two protocols and an IoT device, initialised with different configurations. This is a threat to external validity, in the sense that the results about the component detection and the model accuracy cannot be generalised to all software systems. This is why the experiments deliberately avoid drawing any general conclusion. We chose to mainly concentrate our experimentations on one system that we implemented to be able to appraise the capability of COnfECt of returning correct models. This threat is somewhat mitigated by the fact that we used HTTP traces as inputs, which can be collected from numerous Web applications. In addition, one of the components of the IoT device is a small Web server running a classical Web site. We hence believe that our tool can be easily generalised to Web applications. But, it is manifest that more experimentations are required, on further kinds of systems.

The generalisation of our approach is also restricted by the three hypotheses H1 to H3. In H1, we chose to consider that the internal calls among components are removed within the traces. However, if the synchronisation actions are available in traces, our algorithm may be modified to take them into consideration instead of adding synchronisation actions. We assume that components are not executed in parallel (H2) and that there exists a single root component (H3). With some factors, e.g., $f_1/f_1'$, we could update COnfECt to consider systems having several root components calling other components in parallel. This could be done by identifying every component with the factor $f_1$ and then splitting traces into sub-traces when parallel executions are detected with an analysis of the action timestamps. But, at the moment, this modification depends on the employed factors and cannot be generalised.

There are also several threats to internal validity. Firstly, like all the other model learning approaches using traces, the more the traces, the more complete the models will be. Furthermore, our approach uses similarity factors and thresholds, like the approaches used in machine learning. This kind of approach requires some expertise to choose the right factors and thresholds. In our case, the generation of

accurate models appears to be laborious without having any expertise allowing to adjust the component detection. We indeed observed that an expert is necessary either to provide some information about the components (e.g., means to identify components) or to be able to observe wrong behaviours in the models and to follow the threshold choice protocol we listed in Section 7.1. Conversely, if the model learning is supervised by an expert, COnfECt infers relevant models in reasonable time delays.

## 8 Conclusion

We have presented COnfECt, a model learning method that generates systems of LTSs from execution traces. A system of LTSs captures the behaviours of components and their synchronisations. COnfECt is made up of several algorithms, themselves based on some machine learning techniques to detect components in traces. Additionally, it proposes three LTS synchronisation strategies, which help manage the model generalisation. Learned models are a good mean to ease bug detection (Durand and Salva, 2015; Ohmann et al, 2014; Mariani and Pastore, 2008). As systems of LTSs show how components behave and are synchronised, we believe that these models offer better readability and comprehensibility than those inferred by classical model learning tools for finding and locating bugs. Here a bug can be more precisely located on a LTS and hence on a specific component.

In future work, we firstly intend to perform more evaluations of COnfECt on several kinds of systems. From the lessons learned through this work, it appears that another immediate line of future work is to reduce the requirements of the approach. COnfECt, which uses machine learning techniques, needs to be supervised by an expert of the system in order to infer correct models. We intend to revise the COnfECt algorithm to better integrate this supervision need. For instance, we could help engineers find the parameter assignments used to identify components. Or we could ask them the expected number of components and find the most appropriate factors and thresholds. Another challenge is to get rid of some hypotheses, e.g., the need to collect traces from components having synchronous interactions.

Several approaches, e.g., (Beschastnikh et al, 2011; Ohmann et al, 2014; Beschastnikh et al, 2014) mine temporal invariants from logs to increase the accuracy of the generated models. This technique sounds interesting but cannot be directly applied to COnfECt as we split traces and build several LTSs. We need to study if it is of interest to mine invariants after the trace extraction. A system of LTSs also offers the possibility to derive models having different levels of abstraction, by hiding some components or not. This notion of abstraction sounds interesting and needs more investigations. For instance, bug or security analysis could be focused on some components only with respect to a given risk criterion, while reducing the analysis efforts.

## 9 Acknowledgement

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

**References**

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

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

Ammons G, Bodík R, Larus JR (2002) Mining specifications. SIGPLAN Not 37(1):4–16, DOI 10.1145/565816.503275

Angluin D (1987) Learning regular sets from queries and counterexamples. Information and Computation 75(2):87 – 106

Antunes J, Neves N, Verissimo P (2011) Reverse engineering of protocols from network traces. In: Reverse Engineering (WCRE), 2011 18th Working Conference on, pp 169–178, DOI 10.1109/WCRE.2011.28

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

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

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`

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

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

Cohen WW, Ravikumar P, Fienberg SE (2003) A comparison of string distance metrics for name-matching tasks. In: Proceedings of the 2003 International Conference on Information Integration on the Web, AAAI Press, IIWEB'03, pp 73–78

Dallmeier V, Knopp N, Mallon C, Fraser G, Hack S, Zeller A (2012) Automatically generating test cases for specification mining. IEEE Trans Softw Eng 38(2):243–257, DOI 10.1109/TSE.2011.105

Dupont P (1996) Incremental regular inference. In: Proceedings of the Third ICGI-96, Springer, pp 222–237

Durand W, Salva S (2015) Passive testing of production systems based on model inference. In: ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA,, ACM, Austin,

Texas, USA, pp 138–147

Ernst MD, Cockrell J, Griswold WG, Notkin D (1999) Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '99, pp 213–224

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

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

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

Hangal S, Lam MS (2002) Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '02, pp 291–301, DOI 10.1145/581339.581377

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

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

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

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)

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

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

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

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

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

Meinke K, Sindhu M (2011) Incremental learning-based testing for reactive systems. In: Gogolla M, Wolff B (eds) Tests and Proofs, Lecture Notes in Computer Science, vol 6706, Springer Berlin Heidelberg, pp 134–151, DOI 10.1007/978-3-642-21768-5_11

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

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

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

Petrenko A, Avellaneda F, Groz R, Oriat C (2017) From passive to active fsm inference via checking sequence construction. In: Yevtushenko N, Cavalli AR, Yenigün H (eds) Testing Software and Systems, Springer International Publishing, Cham, pp 126–141

Pradel M, Gross TR (2009) Automatic generation of object usage specifications from large method traces. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, ASE '09, pp 371–382

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

Salva S, Blot E (2018) Confect: An approach to learn models of component-based systems. In: Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018., pp 298–305, DOI 10.5220/0006848302980305

Salva S, Blot E, Laurençot P (2018) Combining model learning and data analysis to generate models of component-based systems. In: Testing Software and Systems - 30th IFIP WG 6.1 International Conference, ICTSS 2018, Cádiz, Spain, October 1-3, 2018, Proceedings, pp 142–148, DOI 10.1007/978-3-319-99927-2_12

Shahbaz M, Groz R (2013) Analysis and testing of black-box component based systems by inferring partial models. Software Testing, Verification and Reliability DOI 10.1002/stvr.1491

Tan PN, Steinbach M, Kumar V (2005) Introduction to Data Mining, (First Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

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

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

Willett P (1988) Recent trends in hierarchic document clustering: a critical review. Information Processing & Management 24(5):577–597

Yoon KP, Hwang CL (1995) Multiple attribute decision making: An introduction (quantitative applications in the social sciences)

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