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 black box system from its execution traces. COnfECt is specialised into the detection of components of a black-box system and in the inference of models called systems of LTSs. For every component discovered, COnfECt generates a Labelled Transition System (LTS), 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 synchronises them. We describe the three steps of COnfECt and the related algorithms in this paper. Then, we present some preliminary experimentations on a real system.

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 overcame with model learning approaches (Angluin, 1987; Biermann and Feldman, 1972; Ernst et al, 1999; Meinke 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

Sébastien Salva (✉)
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

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, surprisingly, little attention has been given to the generation of models from integrated systems made up of components. In the previously cited papers, systems are indeed seen as single units. 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 subparts 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 method called COnfECt (COrrelate Extract Compose) 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. This model encodes the behaviours of every component by a LTS and shows how they are synchronised together. 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 identifying distinctive component behaviours and extracts sub-traces. The last step generates a system of LTSs by means of three strategies. These 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 appraise its benefits. We provide a preliminary evaluation in the paper, which deals with the correct component detection, the relevance and size of the models, and the efficiency/scalability of the algorithm. We also examine potential threats to the validity of our evaluation.

Paper organisation: Section 2 presents some papers related to our approach. Section 3 recalls some definitions about the LTS model. Section 4 describes the steps of the COnfECt approach, illustrated with an example. The next section shows the results of the experimentation of COnfECt and discusses about the threat to validity. Finally, Section 6 summarises our contributions and draws some perspectives for future work.

2 Related Work

Model learning can be defined as *a set of methods that 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) were

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.

After having studied the literature, we have observed that few papers tackled Challenge 1 or 2. Groz et al (2008) proposed to generate a controllable approximation of components through active testing. 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. CSight (Beschastnikh et al, 2014) is another approach that infers CFSM (Communicating FSM) of concurrent systems, exchanging messages through channels. The components are known, and provide separate trace sets. The exchanged messages are observable, including those that synchronise the components together. The approach synchronises the components on the channels used to exchange messages, and refines the models with invariants like Synoptic. Our approach does not require these assumptions. It still uses kTail, but also data analysis techniques to answer to these challenges.

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 (Component 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

We propose to express the behaviours of components with the well established Labelled Transition System (LTS) model. The use of LTSs allows to exploit the definitions related to the LTS composition, for instance given by van der Bijl et al

(2004). The LTS model is defined in terms of states and transitions labelled by actions, taken from a general action set $\mathcal{L}$, which expresses what happens. τ 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 τ 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: $q \xRightarrow{a_1 \ldots a_n} q' =_{def} \exists q0 \ldots q_n, q = q0 \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q_n = q'$. The concatenation of two action sequences σ_1, $\sigma_2 \in \mathcal{L}_\tau^*$ is denoted $\sigma_1.\sigma_2$. ϵ 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 $q0a_1 ... q_{k-1}a_k q_k$ is an alternate sequence of states and actions such that: $\exists q_{i-1}, q_i, a_i, (1 \leq i \leq k) : q0 \xRightarrow{a_1 \ldots a_k} q_k \in \rightarrow^*$. $Runs(L)$ 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 $r = q0a_1 ... q_{k-1}a_k q_k$, 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 shared actions, also called *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 $C = \{C_1, \ldots, C_n\}$ 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 The COnfECt Approach

COnfECt (COrrelate Extract Compose) aims at answering to Challenge 1: how to infer a system of LTSs SC from the traces of SUL, in such a way that 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 indeterministic, uncontrollable or can have cycles among its internal states. However, to answer to this research problem, we assume that SUL obeys certain restrictions:

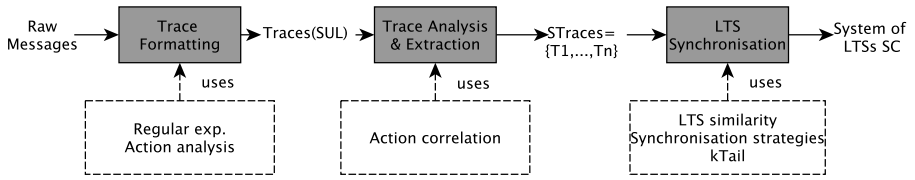


Fig. 1: The COnfECt approach overview

- **H1:** SUL **as a black box.** SUL is a black box including components from which only communications with the environment can be observed. The communications 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 consider that messages are collected in a synchronous manner (by means of synchronous communications) and include timestamps.
- **H3: Single root component** we suppose that the traces of $Traces$(SUL) capture the behaviours of one first component calling other components.

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 detecting 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. These steps are detailed below.

4.1 Trace Formatting

COnfECt takes raw messages that are totally ordered by means of their timestamps. These messages are firstly filtered and formatted by applying regular expressions. We consider that these expressions transform a message into a action of the form $a(\alpha)$ with a a label and α 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.

Then, COnfECt proposes four ways to split a list of actions into traces: by requesting a trace identifier, by inspecting timestamps, or a combination of both. The first mode, proposed by several model learning approaches, combines actions having the same identifier into the same execution trace. The seconds 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 $Traces$(SUL), which gathers traces of the form $a_1(\alpha_1)...a_k(\alpha_k)$.

4.2 Trace Analysis & Extraction

Algorithm 1: Trace refinement Algorithm

input : $Traces(\text{SUL}) = \{\sigma_1, \ldots, \sigma_m\}$
output: $STraces = \{T_1, \ldots, T_n\}$
1 $T_1 = \{\}$;
2 $STraces = \{T_1\}$;
3 **foreach** $\sigma \in Traces$(SUL) **do**
4 $\quad \sigma'_1\sigma'_2 \ldots \sigma'_k$=Inspect($\sigma$);
5 $\quad STraces$=Extract($\sigma'_1\sigma'_2 \ldots \sigma'_k, T_1$);
6 return $STraces$;

This step identifies component behaviours in the traces of $Traces$(SUL), it splits them and returns a set $STraces = \{T_1, \ldots, T_n\}$ such that each trace set T of $STraces$ includes traces of one component. Algorithm 1, which implements this 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 give birth to LTSs. These procedures are explained below.

4.2.1 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 $f_1, \ldots\ f_k$ 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. The factor choice needs to be addressed by an expert of the system SUL. The more he/she has knowledge about it, the more

precise the component detection will be. We give below three factor examples. The first one is based on the component identification. It is here assumed that components are identified with a parameter set and that this set is known. The two last factors are more general and do not require any knowledge about SUL for being used.

- $f_1(a_1(\alpha_1), a_2(\alpha_1))= 1$ iff $Id(\alpha_1) = Id(\alpha_2)$ with $Id(\alpha)$ the assignment in α of the parameters that identify every component. Otherwise, $f_1(a_1(\alpha_1), a_2(\alpha_2))= 0$;
- $f_2(a_1(\alpha_1), a_2(\alpha_2)) = max(\frac{\text{freq}(a_1a_2)}{\text{freq}(a_1)}, \frac{\text{freq}(a_1a_2)}{\text{freq}(a_2)})$ with freq(a_1a_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_1a_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);
- $f_3(a_1(\alpha_1), a_2(\alpha_2)) = sim_j(a_1(\alpha_1), a_2(\alpha_2))$ with sim_j the Jaro Similarity of two strings, which compares two strings on their common characters. Informally, this factor evaluates the similarity of two strings with regard to the character order, number and the shared characters. Other string similarities could be used with regard to the system context. 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(σ_1) holds when σ_1 has successive actions that strongly correlate. We also define the weak correlation of two action sequences. σ_1 $weak-corr$ σ_2 holds when the last event of σ_1 does not strongly correlates with the first one of σ_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(σ_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. σ_1 weak-corr σ_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 σ of $Traces$(SUL) and potentially segments σ into (sub-)sequences such that each sequence σ_1 has a strong correlation and has a weak correlation with the next sequence σ_2. We consider that these distinctive sequences σ_1 σ_2 express the behaviour of two components, a component produces σ_1 and calls a second component, which produces σ_2.

Algorithm 2: Procedures Inspect and Extract

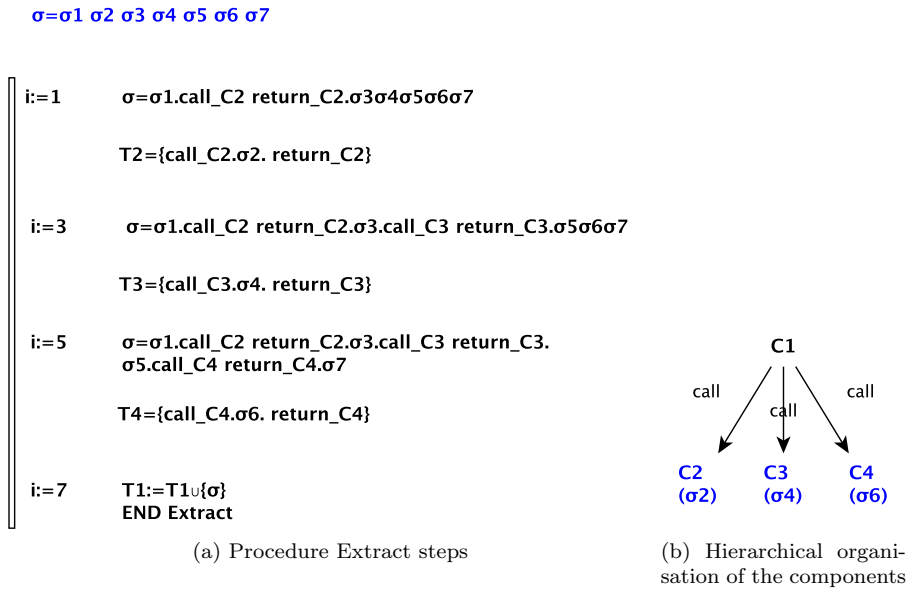
Procedure *Inspect*(σ) : $\sigma'_1\sigma'_2 \dots \sigma'_k$ **is**
  Find the non-empty sequences $\sigma'_1\sigma'_2 \dots \sigma'_k$ such that: $\sigma = \sigma'_1\sigma'_2 \dots \sigma'_k$,
    strong-corr$(\sigma'_i)_{(1 \leq i \leq k)}$, $(\sigma'_i$ weak-corr $\sigma'_{i+1})_{(1 \leq i \leq k-1)}$;
Procedure *Extract*($\sigma = \sigma_1\sigma_2 \dots \sigma_k, T_c, STraces$): $STraces$ **is**
  $i := 1$;
  while $i < k$ **do**
    $n := |STraces| + 1$;
    $T_n := \{\}$;
    $STraces := STraces \cup \{T_n\}$;
    σ_p is the prefix of σ up to σ_i;
    if $\exists j > i$: *strong-corr*$(\sigma_i.\sigma_j)$ **then**
      σ_j is the first sequence in $\sigma_i \dots \sigma_k$ such that strong-corr$(\sigma_i\sigma_j)$;
      $\sigma := \sigma_p\sigma_i.call\_C_n return\_C_n.\sigma_j \dots \sigma_k$;
      if $(j - i) > 2$ **then**
        Extract$(\sigma_{i+1} \dots \sigma_{j-1}, T_n)$;
      else
        $T_n := T_n \cup \{call\_C_n.\sigma_{i+1}.return\_C_n\}$;
      $i := j$;
    else
      $\sigma := \sigma_p\sigma_i.call\_C_n return\_C_n$;
      if $(k - i) > 1$ **then**
        Extract$(\sigma_{i+1} \dots \sigma_k, T_n)$;
      else
        $T_n := T_n \cup \{call\_C_n.\sigma_k.return\_C_n\}$;
      $i := k$;
  if $c \neq 1$ **then**
    $\sigma = call\_C_c.\sigma.return\_C_c$;
  $T_c := T_c \cup \{\sigma\}$;
  return $STraces$;

/devices() **/json.htm(idx:=115,svalue:=15.00) Response(status:=200)** Response(status:=200,data:=[1]) **/json.htm(idx:=115,svalue:=16.00) Response(status:=200)** /devices() Response(status:=200,data:=[1]) /hardware() Response(status:=200,data:=[2]) /config() **/json.htm (idx:=0,switchcmd:=On) Response(status:=200)** Response(status:=200, data:=[2]) /tools() Response(status:=200,data:=[3])

Fig. 2: Example of a formatted trace collected from an IoT device. The sequences in bold are strongly correlated and weakly correlated to the others.

Example 1 The trace of Figure 2 stems from the monitoring of a real smart thermostat device providing traces at the HTTP level. This trace, composed of 16 actions, was formatted with 4 regular expressions, which assign the called URL and the HTTP responses to labels, and keep some data, e.g., the temperature with the parameter svalue.

Let us illustrate the trace analysis step on a trace set $Traces$(SUL) including the trace of Figure 2, and with a Correlation coefficient defined with the factor f_2, which computes the frequency of having successive actions in traces. The use of this Correlation coefficient reveals that the trace of Figure 2 has 3 sub-sequences (in bold in the figure), which are frequently encountered and which are not correlated to the other actions. The trace is then arranged into seven sub-sequences.



(a) Procedure Extract steps

(b) Hierarchical organisation of the components

Fig. 3: Sequence extraction example

4.2.2 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 σ, splits it and stores the resulting trace into a set T_c. Given a sequence σ_i of the trace $\sigma = \sigma_1 \dots \sigma_k$, the procedure *Extract* tries to find the next sequence σ_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 σ_j is not found) is extracted as it exposes the behaviour of other components that are called by the current one. If this sequence σ' is not composed of more than two sub-sequences, 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 σ, the sequence σ' is removed and replaced by the actions $call\_C_n.return\_C_n$. After the covering of every sub-sequence of σ, the procedure Extract eventually checks whether σ 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 σ is surrounded with $call\_C_c$ and $return\_C_c$ to express that σ stems from a component that was previously called by another one. Otherwise, the sequence σ remains unchanged.

Example 2 Let us illustrate the functioning of the procedure Extract with the example of Figure 3a, which takes back the trace of Figure 2. This trace was segmented into seven parts. We start at σ_1 (i:=1). The first sequence that is strongly

```
STraces = {
T1 {/devices() call_C2 return_C2 Response(status:=200,data:=[1])
call_C3 return_C3 /devices() Response(status:=200,data:=[1])
/hardware() Response(status:=200,data:=[2]) /config() call_C4
return\_C4 Response(status:=200,data:=[2]) /tools()
Response(status:=200,data:=[3])}
T2 {call_C2 /json.htm(idx:=115,svalue:=15.00) Response(status:=200)
return_C2}
T3 {call_C3 /json.htm(idx:=115,svalue:=16.00) Response(status:=200)
return_C3}
T4 {call_C4 /json.htm(idx:=0,switchcmd:=On) Response(status:=200)
return_C4} }
```

Fig. 4: Example of formatted trace segmented into 4 trace sets and completed with synchronisation actions.

correlated with σ_1 is σ_3. The sequence σ_2 is extracted and replaced by the actions $call\_C2$ $return\_C2$. The procedure is not recursively called as σ_2 is not composed of several weakly correlated action sequences. The sequence σ_2 is now surrounded with the actions $call\_C2$ and $return\_C2$ to prepare the LTS synchronisation. The resulting sequence is added to the set T_2. We go back to the trace σ at the subsequence σ_3 (i:=3). The same process is applied on σ_4 and later on σ_6 until the algorithm reaches the end of the sequence σ (with i:=7). The trace σ 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 σ comes from $Traces$(SUL), which means that σ captures the behaviour of a component that has not been called by another component. Hence, this trace is not surrounded by synchronisation actions. σ is placed into the trace set T_1. At the end of this process, we have recovered the hierarchical structure of components depicted in Figure 3b. And we get four trace sets, given in Figure 4.

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).

4.3 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.

As regards the hypothesis H2, 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.

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

Algorithm 3: LTS synchronisation strategies

input : System of LTSs $SC = \langle S, C \rangle$ with $C = \{C_1, \ldots, C_n\}$, strategy
output: System of LTSs $SC_f = \langle S_f, C_f \rangle$
if *strategy = Strict Synchronisation* **then**
  return $kTail(k = 2, SC)$;
else
  $\forall (C_1, C_2) \in C^2$ Compute $Similarity_{LTS}(C_1, C_2)$;
  Build a similarity matrix;
  Group the LTSs into clusters $\{Cl_1, \ldots Cl_k\}$ such that $\forall (C_1, C_2) \in Cl_i^2 : C_1$ similar $C2$;
  foreach *cluster* $Cl = \{C_1, \ldots, C_l\}$ **do**
    C_{Cl}:=Disjoint Union of the LTSs $C_1, \ldots, C_l$;
    $C_f = C_f \cup \{C_{Cl}\}$;
  foreach $C_i = \langle Q, q0, \Sigma, \rightarrow \rangle \in SC_f$ **do**
    foreach $q_1 \xrightarrow{a} q_2$ *with* $a = call\_C_m$ *or* $a = return\_C_m$ **do**
      Find the Cluster Cl such that $C_m \in Cl$;
      Replace C_m by C_{Cl} in the label a;
      $S_f := S_f \cup \{a\}$;
    foreach $q_1 \xrightarrow{call\_C_m return\_C_m} q_2 \in \rightarrow$ **do**
      Merge (q_1, q_2);
  if *strategy = Strong Synchronisation* **then**
    foreach $C_i = \langle Q, q0, \Sigma, \rightarrow \rangle \in C_f$ **do**
      Complete the outgoing transitions of the states of Q so that C_i is callable-complete;
      foreach $q_1 \xrightarrow{call\_C_m return\_C_m} q_2 \in \rightarrow$ **do**
        Merge (q_1, q_2);
  return $kTail(k = 2, SC_f)$

4.3.1 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 $C = \{C_1, \ldots, C_n\}$. QF is the set of final states of the LTS $C_1 \parallel C_2 \parallel \cdots \parallel C_n$.*
$Traces_{QF}(SC) = Traces$(SUL).

4.3.2 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 $f'_1, \ldots f'_k$ 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 α 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.

4.3.3 Strong Synchronisation:

This strategy aims at providing more general models than the Weak strategy, by returning callable-complete LTSs. We say that a LTS C_1 of a system of LTSs SC is callable-complete when C_1 can call any LTS C_2 of SC at any of its states:

Definition 8 (Callable-complete LTS) Let $SC = \langle S, C \rangle$ be a system of LTSs. A LTS $C_1 = \langle Q_1, q0_1, \Sigma_1, \rightarrow_1 \rangle \in C$ is said callable-complete over SC iff $\forall q \in Q_1, \forall C_2 \in C \setminus \{C_1\}, \exists q' \in Q_1 : q \xRightarrow{call\_C_2.return\_C_2} q'$.

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 over SC.

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

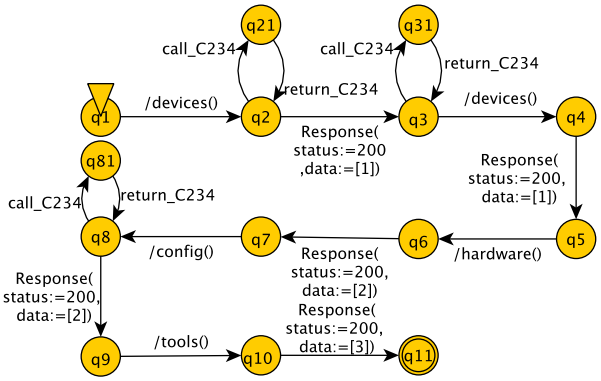
Proposition 2 *Let $SC = \langle S, C \rangle$ be a system of LTSs achieved with the Weak or Strong synchronisation strategy (before the call of kTail), with $C = \{C_1, \ldots, C_n\}$. QF is the set of final states of $C_1 \parallel C_2 \parallel \cdots \parallel C_n$.*
$Traces_{QF}(SC) \supset Traces(\text{SUL})$.

Finally, 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. We use here 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).

Example 3 We illustrate this step with the set $STraces$ of Figure 4 and with the Weak strategy. Each trace set is firstly transformed into a LTS. As the trace sets of Figure 4 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 5a 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. Figures 5b and 6a depict the resulting LTSs C_1 and C_{234}. As we choose the Weak synchronisation strategy, the transition sequences of the form

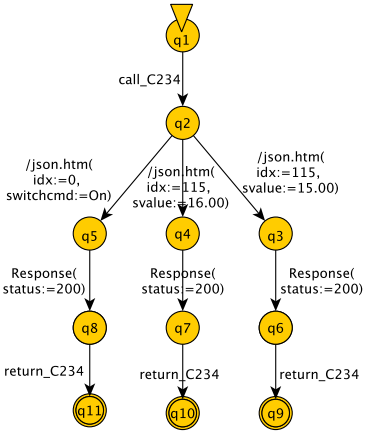
	C_1	C_2	C_3	C_4
C_1	1	0	0	0
C_2	0	1	0.5	0,5
C_3	0	0.5	1	0,5
C_4	0	0,5	0,5	1

(a) LTS Similarity matrix

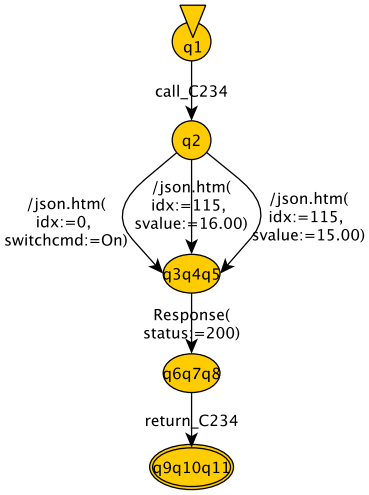


(b) LTS C_1 modelling the Web interface

Fig. 5



(a) LTS C_2_3_4 modelling the sensors



(b) Reduced LTS C_2_3_4 after kTail

Fig. 6

$q_1 \xRightarrow{call\_C_m return\_C_m} q_2$ have been replaced with loops in these LTSs. From the traces of Figure 4, we finally obtain two LTSs: 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 $(q9, q10, q11)$. kTail merges them and returns the LTS of Figure 6b.

5 Preliminary Evaluation

We have implemented COnfECt in a prototype tool[1], which takes raw messages and generates systems of LTSs. The first step of COnfECt, which performs the trace formatting by means of regular expressions and automatically assembles actions into traces, is implemented in first a tool called *TFormat*[2]. For the LTS synchronisation step, 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 number of clusters.

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?
- C2 (Relevance of the models): is COnfECt able to infer models that accept correct behaviours, including new traces not used for the model generation? Is COnfECt able to infer concise and readable models?
- C3 (Efficiency/Scalability): how long does COnfECt take to generate systems of LTSs? Can COnfECt take large trace sets?

5.1 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 an IoT device, whose source code is available here[3]. This device is a smart connected thermostat controlling heat pumps, which 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. We ran the IoT device with several component configurations: one different component is started in Conf. 1 to 3, two components are loaded in Conf. 3 to 6, and 3 components in Conf. 7 to 10. The HTTP traces were formatted with our tool TFormat and 10 regular expressions. These traces have the same form as the trace given in Example 2. The tool and the trace sets are available here[4]. The LTS generation was performed on a desktop computer with 1 Intel(R) CPU i5-7500 @ 3.4GHz and 16GB RAM.

[1] https://github.com/Elblot/COnfECt
[2] https://github.com/sasa27/TFormat
[3] https://github.com/sasa27/OpenThermostat
[4] https://github.com/Elblot/COnfECt

5.1.1 Factor choice & thresholds assessment

The Correlation and LTS Similarity coefficients have to be defined by setting factors, weights and thresholds. We chose for the experiments the factor combinations f_1/f_1' and f_2/f_2'.

The factor combination f_2/f_2', which is based on the labels found in traces, does not require any expert knowledge to generate models. But it is manifest that the choice of the thresholds has a strong influence on the accuracy of the models. An expert is hence required to appraise this accuracy and the thresholds. For the experiments, we applied this protocol:

1. generation of the first models with the 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;
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.

We followed this protocol with the three configurations Conf. 7 to 9. 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 used this factor combination in Conf. 10.

5.2 C1 (Component detection)

Configuration	# real Components	Factors	Strict	Weak	Strong
Conf. 1	1	$f_2 \geq 0.5; f_2' \geq 0.75$	10	1	1
Conf. 2	1	$f_2 \geq 0.4; f_2' \geq 0.75$	1	1	1
Conf. 3	1	$f_2 \geq 0.4; f_2' \geq 0.75$	1	1	1
Conf. 4	2	$f_2 \geq 0.4; f_2' \geq 0.75$	85	2	2
Conf. 5	2	$f_2 \geq 0.4; f_2' \geq 0.75$	105	2	2
Conf. 6	2	$f_2 \geq 0.4; f_2' \geq 0.75$	67	2	2
Conf. 7	3	$f_2 \geq 0.75; f_2' \geq 1$	345	345	345
Conf. 8	3	$f_2 \geq 0.6; f_2' \geq 1$	181	181	181
Conf. 9	3	$f_2 \geq 0.6; f_2' \geq 0,75$	181	3	3
Conf. 10	3	$f_1 \geq 1; f_1' \geq 1$	117	3	3

Table 1: Number of components detected by COnfECt.

With every configuration, we collected and formatted a set of 10 traces composed of about 50 actions. 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. 2-6, 9,10 show the results achieved with COnfECt when the thresholds X and Y are correctly set. The approach detects a correct number

of components whatever the strategy used in Conf. 2 and 3. With Conf. 4-6, 9 and 10, the Strict strategy provides too much 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.

In Conf. 1, we observe the generation of 10 LTSs with the Strict strategy instead of having one LTS. Here, the threshold X was not appropriate and involved the segmentation of traces. The inappropriate threshold X is implicitly corrected with the Weak and Strong strategies because a correct Similarity threshold is given. As a result, these two strategies, which merge the similar LTSs, return one LTS as expected in Conf. 1.

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.

With Conf. 10, the trace segmentation and the LTS similarity is based on the component identification (factors f_1/f_1'). With this configuration, the number of components is correctly detected with the Weak and Strong strategies, without the need for adjusting thresholds like with f_2/f_2'.

We manually analysed the LTSs built with the configurations and strategies giving a correct number of components, to check whether each expresses the behaviours of only one real component. We did not observe any mixture of behaviours. These experiments show that COnfECt answers to Challenge 1, when the factors and the thresholds are correctly set. The general functioning of COnfECt is illustrated in Conf. 2-6, 9 and 10: the Strict strategy refines the traces and often returns to much LTSs. The two last strategies counterbalance the trace refinement.

5.3 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' only take two values each. But, f_2/f_2' have to be evaluated with several model generation attempts.

We analysed the models generated by COnfECt to measure their ability in accepting correct behaviours, i.e. traces collected from the system under learning, which do not necessarily belong to the trace set used for the model generation. As the model quality depends on the methods, the coefficient thresholds and strategies, we chose to take back the 20 models produced by kTail and COnfECt with the configurations Conf. 2-6, 9,10. The models capture the behaviours of two or more components and are built with correct thresholds. We collected from the IoT device a set of 20 traces (of about 50 actions each) with Conf. 2 to 6, and 50 traces with Conf. 9,10, which aim at covering at least one time the actions of the IoT device. In comparison to the traces used to build models, we observed that they may be composed of repetitive actions, of calls of components at different internal

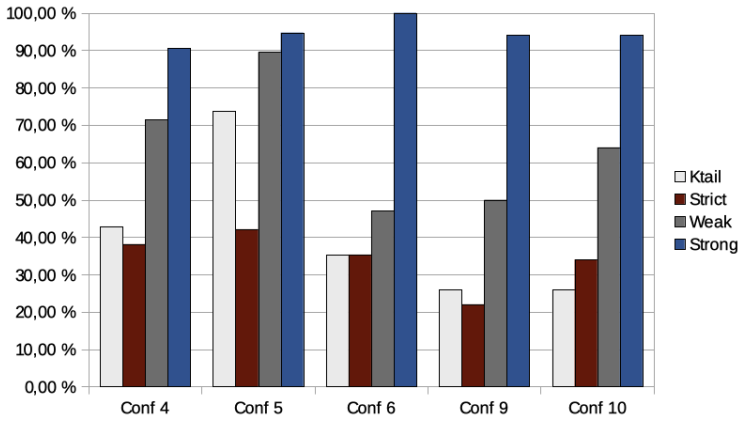


Fig. 7: Rates of traces accepted by models.

states of the IoT device, or of some new actions, e.g., new URL requests. But, these are not composed of calls of unknown components.

Figure 7 illustrates the rates of accepted traces by configuration and models generated by kTail and the three strategies of COnfECt. The models inferred by kTail and the Strict strategy give close results, except in Conf. 5. We studied the two models and deduced that the system of LTSs given by the Strict strategy splits the traces into several LTSs in which less states have been merged in comparison to the number of states merged in the LTS given by kTail. As a consequence, the system of LTSs is less general. This is why it rejects more traces. Whatever the configuration, the models inferred by the two last strategies of COnfECt accept more correct behaviours than the models of kTail. This increase of accepted traces is a consequence of allowing repetitive component calls in the LTSs. Unsurprisingly, the Strong strategy provides the models that give the highest rates of accepted traces (between 90 and 100 %). The LTSs are here callable-complete and encode the call of any component from any state. After having studied the models given by the Strong strategy, we deduced that it provides more general models that are correct in the context of IoT devices. Indeed, this strategy perfectly matches the functioning of this kind of systems, which are composed of components that can be repetitively executed. But, with other systems, this strategy could infer models that accept incorrect behaviours. In general terms, the Weak strategy sounds to be a good compromise. But, it also seems interesting to investigate whether more strategies tailored for specific kinds of systems could be defined.

We evaluated the readability of the models generated by COnfECt and kTail by measuring the model sizes in Conf. 1 to 10. The first four 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 (excepted with Conf. 2 and 3 since there is only one component). This outcome stems from our algorithm, which adds transitions labelled by synchronisation actions. With the Strict strategy, the state number is increased by 886 % 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

Configuration	kTail		Strict		Weak		Strong		Strict+hide		Weak+hide		Strong+hide	
	#states	#trans	#states	#trans	#states	#trans	#states	#trans	#states	#trans	#states	#trans	#states	#trans
Conf. 1	40	66	152	169	46	78	60	150	120	137	39	70	36	67
Conf. 2	6	8	6	8	6	8	6	8	6	8	6	8	6	8
Conf. 3	9	16	9	16	9	16	9	16	9	16	9	16	9	16
Conf. 4	60	115	731	691	104	188	72	183	399	359	71	124	36	85
Conf. 5	65	125	798	752	105	200	71	178	397	349	69	128	34	74
Conf. 6	22	47	496	470	41	81	25	57	236	210	24	55	10	23
Conf. 9	85	175	1307	1185	158	286	82	197	627	505	96	169	36	87
Conf. 10	85	175	915	908	169	312	126	245	491	484	92	163	38	64

Table 2: Sizes of the LTSs obtained with kTail and the three strategies of COnfECt. "hide" refers to the removal of the LTS transitions labelled by synchronisation actions.

and should be restricted to small trace sets only. The state number is increased by 52 % and 17 % with the Weak and Strong strategies.

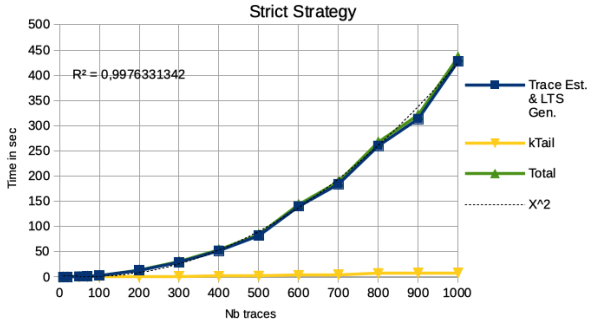
The transition labelled by synchronisation actions help interpret the component combination and are required to later compose LTSs. But, these are not significant if one want to focus on the component behaviours only. Table 2 provides, in the last three columns, the number of states and transitions after applying the hide operation, which removes the transitions labelled by the synchronisation actions. The models generated by COnfECt become more concise than those obtained with kTail. More precisely, we obtain about the same numbers of states/transitions with the Weak+hide strategy and kTail. But the former divides the system behaviours into several smaller LTSs, which are much more readable. The state numbers are reduced by 33 % when using the Strong+hide strategy. For instance, the number of states is equal to 36 in Exp. 7, whereas the LTS achieved with kTail has 85 states.

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. The models built by the Strict strategy and kTail reject a similar ratio of correct behaviours. The Weak and Strong strategies of COnfECt outperform kTail in terms of generation of more general and correct models. Then, we showed that COnfECt builds larger models. But, if we hide the synchronisation actions, the systems of LTSs are more concise than the LTSs achieved by kTail (with the Weak and Strong strategies). It is also worth to mention that the systems of LTSs, which split the component behaviours, are more readable than big LTSs.

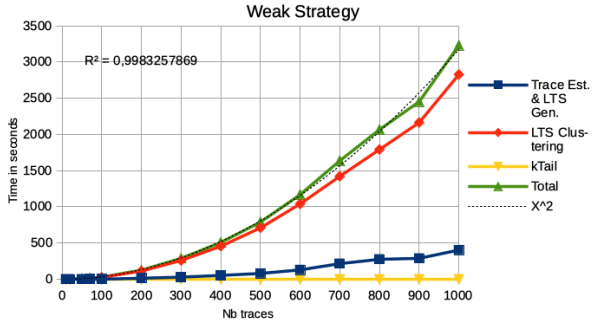
5.4 C3 (Efficiency/Scalability)

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. 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.

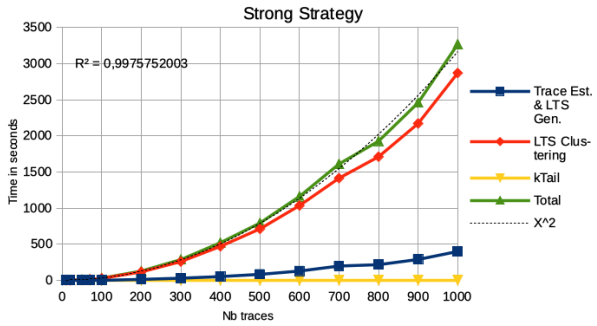
With the Strict strategy and trace sets having no more than 100 traces (10, 20, 50, 100), COnfECt builds systems of LTSs in less than 3 seconds. We observed that the evaluation of the factor f_2 takes most of the time as the action set needs to be scanned with two nested loops. Hence, its is not surprising to observe



(a)



(b)



(c)

Fig. 8: Execution times vs. nb of traces

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.

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.

5.5 Threat to Validity

There are many application and system contexts, but this preliminary experimental evaluation is only applied on 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 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. We observed that this is usually the case as monitors usually cannot observe internal communications among components. But, if the synchronisation actions are available in traces, our algorithm may be modified to take them into consideration instead of adding synchronisation actions. With H2/H3, we assume that components are not executed in parallel and that there exists a single root component. With some factors, e.g., f_1/f_1', we could update COnfECt to consider systems having several root components calling other components. 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 5.1. Conversely, if the model learning is supervised by an expert, COnfECt infers relevant models in reasonable time delays.

We thus believe that this preliminary experience provides relevant insights on the benefits of using our tool.

6 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). 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.

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

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

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

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

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)