Reverse Engineering Models of Concurrent Communicating Systems From Event Logs

Sébastien Salva

Reverse Engineering Models of Concurrent Communicating Systems From Event Logs

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

***Abstract*—This paper tackles the problem of recovering formal models of communicating systems made up of components concurrently interacting with each other, e.g., Web service compositions or IoT systems. We present a passive model learning approach, which recovers, from event logs, one Input Output Labelled Transition System (IOLTS) for every component of the system under learning. From an event log, our approach is able to automatically recover conversations (a.k.a. sessions), without having any knowledge about the used event correlation mechanisms. It uses correlation pattern definitions and a heuristic based on the quality of the generated conversations to get the most relevant conversation sets. Then, our approach extracts the trace sets of every component and generates IOLTSs. The latter can be used as documentation, for test case generation, or for formal verification.**

***Keywords*-Reverse engineering; Model learning; Event Log; Communicating systems; .**

I. Introduction

Software Reverse Engineering gathers numerous techniques specialised in the analysis of software system to extract design and implementation information. Among them, model learning has emerged as a highly effective technique for recovering the models of black-box software systems. Such models, e.g., temporal rules, or finite state machines that encode functional behaviours, offer substantial benefits as they can be employed for security audits [1], [2], real-time anomaly detection [3], or bug detection [4].

This paper addresses passive model learning, for which it is assumed that event logs have been previously collected from a system under learning SUL and can be mined to learn models. Although numerous passive model learning algorithms and tools are available in the literature, few of them [5], [4], [6], [7] are directly applicable to distributed systems made up of communicating components. These systems indeed raise specific difficulties. Most of them come from the fact that SUL is made up of components that run in parallel and concurrently interact with each other. To recover the behaviours of the components, it is required to extract accurate conversations (a.k.a. sessions), i.e. event sequences of correlated events interchanged among different components that achieve a certain goal. Additionally, traditional model learning algorithms return "flat" models, i.e. one model for a given composition encoding all the event details (parameters) listed in the event logs. With large event logs, it often results in complex and unreadable models.

Contribution: the paper presents another passive model learning approach, which recovers Input Output Labelled Transition Systems (IOLTSs) from event logs. As SUL is a distributed and concurrent communicating system, we assume that correlation mechanisms, e.g., execution trace identifiers, are employed to propagate context ids and keep track of the process contexts. But, we do not assume knowing how events are correlated in advance. The major contribution of this approach is its capability to automatically retrieve conversations from event logs, without having any knowledge about the used correlation mechanisms. Instead of using a brute-force search over the space of parameter assignments found in events, our algorithm is based upon a formalisation of the notion of correlation patterns and is guided towards the most relevant conversation sets by evaluating conversation quality. As there is no consensus about what a relevant conversation should be, the conversation quality can be adapted to meet user needs and viewpoints. Next, from the retrieved conversations, our approach extracts the trace sets of every component participating in the generation of the event log. And, finally, it generates one IOLTS for every of these components, which captures the behaviours encoded in event logs with inputs and outputs showing the messages received and sent among the components.

The paper is organized as follows: Section II provides some definitions and notations on events, correlations and sessions. Our approach is presented in Section III. Section IV discusses related work. Section V summarises our contributions and draws some perspectives for future work.

II. Events, Correlations and Conversations

A. Preliminary Definitions

We denote $\mathcal{E}$ the set of events of the form $e(\alpha)$ with e a label and α an assignment of parameters in P. The concatenation of two event sequences $\sigma_1, \sigma_2 \in \mathcal{E}^*$ is denoted $\sigma_1.\sigma_2$. ϵ denotes the empty sequence. For sake of readability, we also write $\sigma_1 \in \sigma_2$ when σ_1 is a (ordered) subsequence of the sequence σ_2. Events are partially ordered in event logs. This is expressed with these partial order relations:

- $<_t \subseteq \mathcal{E} \times \mathcal{E}$, which orders two actions according to their timestamps,
- $<_c \subseteq \mathcal{E} \times \mathcal{E}$, which orders two actions if the occurrence of the first action implies the occurrence of the second one,
- $<:=<_t \cup <_c$ is the transitive closure of $<_c$ and $<_t$.

We also use the following notations on events to make our algorithms more readable:

- $from(e(\alpha)) = c$ denotes the source of the event when available; $to(e(\alpha)) = c$ denotes the destination;
- $isReq(e(\alpha))$, $isResp(e(\alpha))$ are boolean expressions expressing the nature of the event;
- $A(\sigma) = \bigcup_{e(\alpha) \in \sigma} \alpha$ is the set of parameter assignments of σ.

In the paper, we use the IOLTS model to express the behaviours of components. This model is defined in terms of states and transitions labelled by input or output events in $\mathcal{E}$.

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

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

B. Event Correlation and Conversations

The correlation mechanisms used from one system to another are seldom the same, but they are often compliant with some patterns. Most of these are introduced in [8]. Given an event sequence $\sigma = e_1(\alpha_1) \dots e_k(\alpha_k) \in \mathcal{E}^*$, we formulate that an event $e(\alpha)$ correlates with σ w.r.t. one of these patterns as follows:

- **Key based correlation:** $e(\alpha)$ correlates with σ if all the events share share the same parameter assignment set: $\alpha \cap \alpha_1 \cap \dots \cap \alpha_k \neq \emptyset$;
- **Chained correlation:** $e(\alpha)$ is correlated with σ if $e(\alpha)$ shares some references with $e_k(\alpha_k)$: $\alpha \cap \alpha_k \neq \emptyset$;
- **Function based correlation:** a function $f : \mathcal{E} \rightarrow L$ firstly assigns to each event a label of the form "l:=label" in L according to the event parameter assignments. Then, the event correlation is performed w.r.t. one of the previous patterns;
- **Time-based correlation:** this pattern is somehow a special case of the previous one, in the sense that a label can be injected into an event w.r.t. a condition on time. A function $f : \mathcal{E} \rightarrow L$ assigns labels to events returns a label of the form "t:=l" according to timestamps.

The above patterns can also be combined with conjunctions or disjunctions to formulate correlation expressions. To make our algorithm readable, we write $e(\alpha)$ *correlates* σ if the event $e(\alpha)$ correlates with a sequence σ by such a correlation pattern-based expression.

```
/login(from:="cl", to:="ShopS", id:="tocken",
    account:="l")
ok(from:="ShopS", to:="cl", id:="tocken"
    trans:="t1")
/order(from:="cl", to:="ShopS",
    trans:="t1",item:="a")
/stock(from:="ShopS", to:="StockS", trans:="t1",
    item:="a")
ok(from:="StockS", to:="ShopS", trans:="t1",
    item:="a")
ok(from:="ShopS", to:="cl",
    trans:="t1",content:="stock")
/supply(from:="ShopS", to:="WS", trans:="t1",
    key:="k1",item:="a")
ok(from:="WS", to:="ShopS", trans:="t1",
    key:="k1")
/supplyWS(from:="WS", to:="WS1", key:="k1",
    key2:="k2",item:="a")
/supplyWS(from:="WS", to:="WS2", key:="k1",
    key2:="k3",item:="a")
/supplyWS(from:="WS", to:="WS3", key:="k1",
    key2:="k4",item:="a")
ok(from:=WS1,to:="WS", key:="k1", key2:="k2")
ok(from:=WS2,to:="WS", key:="k1", key2:="k3")
Unavailable(from:="WS3",to:="WS", key:="k1",
    key2:="k4")
/login(from:="cl", to:="ShopS", id:="tocken2",
    account:="l2")
ok(from:="ShopS", to:=cl, id:="tocken2",
    trans:="t2")
/order(from:="cl", to:="ShopS",
    trans:="t2",item:="b")
ok(from:="ShopS", to:="cl", trans:="t2"
    content:="no stock")
```

Figure 1. Formatted part of an event log

In reference to [9], a set of parameter assignments used for an event correlation is called *correlation set*. A conversation corresponds to an event sequence interchanged among components, whose events correlate by means of correlation sets:

Definition 2 *Let $\sigma = e_1(\alpha_1) \dots e_k(\alpha_k) \in \mathcal{E}^k$.*

- *σ is a conversation iff $\forall 1 < i \leq k : e_i(\alpha_i)$ correlates $e_1(\alpha_1) \dots e_{i-1}(\alpha_{i-1})$*
- *$corr(\sigma) = \{cs_1, \dots, cs_{k-1}\}$ denotes the set of correlation sets of σ, with $cs_i \subseteq (\alpha_i \cup \alpha_{i+1})$*

III. Model Learning

Given an event log produced by a concurrent and distributed system SUL, our approach aims at analysing an event log collected from SUL and at recovering one IOLTS for every component of SUL, which captures its behaviours.

We assume that the events in the event log are ordered with the $<$ relation. When several log files are given, we

C = {
/login() ok() /order() /stock() ok() ok()/supply() ok() /supplyWS()/supplyWS() /supplyWS() ok() ok() unavailable(),
/login() ok() /order() ok()
}
T(SS)={ ?/login() !ok() ?/order() !/stock() ?ok() !ok() !/supply() ?ok(), ?/login() !ok() ?/order() !ok() }
T(WS)={ ?/supply() !ok() !/supplyWS() !/supplyWS() !/supplyWS() ?ok() ?ok() ?unavailable() }
T(St)={ ?/stock() !ok() }
T(WS1)=T(WS2)={ ?/supplyWS() !ok() }
T(WS3)={ ?/supplyWS() !unavailable() }

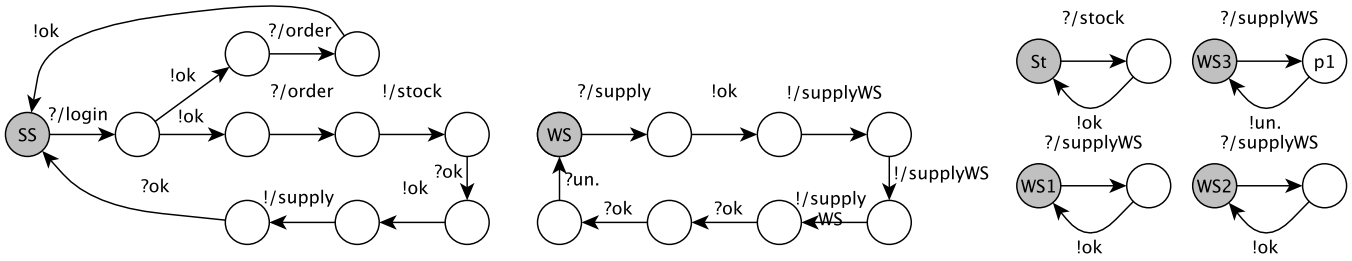

Figure 2. Conversation set and IOLTSs modelling a composition of 6 Web services

assume that they can be assembled with $<_t$ or $<_c$. In particular, the causal order relation $<_c$ may help assemble two log files given by two systems whose internal clock values slightly differ. $<_c$ indeed helps order the actions $a_1(\alpha_1)$ in a first log that imply the occurrence of other actions $a_2(\alpha_2)$ in a second one. The analysis of the pairs $(a_1(\alpha_1), a_2(\alpha_2))$ helps compute the difference of time between these two systems.

Our approach is mainly divided into three main steps: *Conversation extraction*, *Component trace extraction from conversations*, and *IOLTS Generation and Generalisation*. Beforehand, we assume that the event log is formatted into a sequence S of events of the form $e(\alpha)$ by means of regular expressions. The techniques proposed in [4], [10], [11], [12], [13], [14] can assist users in the mining of patterns or expressions from log files, which can be used to quickly derive the appropriate regular expressions.

A. Step 1: Conversation Extraction

The first step of our approach extracts conversations from a sequence S. Our conversation extraction algorithm is devised to explore the possible correlations among the successive events of S, thus in a depth-wise way, while being efficiently guided by the conversation consistencies. This notion of consistency is expressed by means of conversation invariants and conversation quality. Invariants and quality metrics also formulate a heuristic that guides our algorithm towards the most relevant conversation sets.

1) Conversation Invariant and Quality: The correlation patterns implicitly restrict the structure of a conversation according to some properties that are always true, i.e. invariants, over correlation sets. Indeed, an event must correlate with only one conversation σ of a conversation set C with a unique correlation set; a correlation set cs of $corr(\sigma)$ cannot be empty, cs cannot be found in another conversation σ_2 of C. Besides, σ must have parameter assignments for building potential correlation sets, it must include parameter assignments that cannot be found in any other conversation σ_2. These three invariants are formulated in the following proposition. Other invariants can also be added to meet user preferences. For instance, the last invariant imposes conversations to start with a request.

Proposition 3 (Conversation Set Invariants) *Let C be a conversation set and $\sigma \in C$. Inv stands for the set of conversation set invariants:*

- $\forall cs \in corr(\sigma) : cs \neq \emptyset$
- $\forall cs \in corr(\sigma), \forall \sigma_2 \in C \setminus \{\sigma\} : cs \cap A(\sigma_2) = \emptyset$
- $A(\sigma) \setminus \bigcup_{\sigma_2 \in C \setminus \{\sigma\}} A(\sigma_2) \neq \emptyset$
- $\forall e_1(\alpha_1) \dots e_k(\alpha_k) \in C : isReq(e_1(\alpha_1))$

For readability, we denote that the conversations of a conversation set C meet conversation invariants with C *satisfies* Inv.

Our algorithm uses quality metrics as another way to limit the conversation set exploration, but also to prioritize this exploration among several conversation set candidates. We formulate a comprehensive quality metric of a conversation set C by means of a utility function for representing user preferences. The following definition refers to quality metrics $M_i(C)$ over conversation sets, themselves calculated by means of metrics $m_i(\sigma)$ over conversations:

Definition 4 (Conversation Set Quality) $0 \leq Q(C) = \sum_{i=1}^{n} M_i(C).w_i \leq 1$ *with* $0 \leq M_i(C) = \frac{\sum_{\sigma \in C} m_i(\sigma)}{|C|} \leq 1$, $w_i \in \mathbb{R}_0^+$ *and* $\sum_{i=1}^{k} w_i = 1$.

The conversation quality metrics can be general or established with regard to a specific system context. Our approach actually does not limit the metric set. We provide four metric examples below. m_1 and m_2 evaluate whether

a conversation σ follows the classical request-response exchange pattern (sender sends a request to receiver, ultimately returning a response). m_1 evaluates the ratio of requests in σ associated with some responses with $ReqwResp(\sigma)$. m_2 evaluates the ratio of responses following a prior request with $RespwReq(\sigma)$.

$$0 < m_1(\sigma) = \frac{|ReqwResp(\sigma)| + 1}{|Req(\sigma)| + 1} \leq 1 \quad (1)$$

$$0 < m_2(\sigma) = \frac{|RespwReq(\sigma)| + 1}{|Resp(\sigma)| + 1} \leq 1 \quad (2)$$

The metric m_3 examines whether σ is composed of correlated events, in other terms, whether σ has more than one event:

$$m_3(\sigma) = \begin{cases} 1 & \text{if } corr(\sigma) \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

The metric m_4 evaluates the ratio of assignments used to correlate the events of a conversation. The simpler the correlation mechanism is, the closer to 1 the metric is.

$$0 \leq m_4(\sigma) = 1 - \frac{|\bigcup_{cs \in corr(\sigma)} cs|}{|A(\sigma)|} < 1 \quad (4)$$

Algorithm 1: Conversation and Correlation Set Extraction

```
   input : Event sequence S = e_1(α_1) ... e_k(α_k), boolean first
   output: Conversation sets C_1, ..., C_n,
1  C := {e_1(α_1)};
2  call Find_C&CS(C, 2);
3  Procedure Find_C&CS(C, i) is
4      if i ≤ k then
5          foreach σ ∈ C : e_i(α_i) correlates σ do
6              CS := P(α_i ∩ last(σ)) \ {∅} ;
7              foreach cs ∈ CS do
8                  σ' := σ.e_i(α_i);
9                  corr(σ') := corr(σ) ∪ {cs};
10                 C_2 := C ∪ {σ'} \ {σ};
11                 if C_2 satisfies Inv and Q(C_2) ≥ T then
12                     Find_C&CS(C_2, i + 1);
13         C_3 := C ∪ {e_i(α_i)};
14         corr(e_i(α_i)) := ;
15         if C_3 satifies Inv and Q(C_3) ≥ T then
16             Find_C&CS(C_3, i + 1);
17     else
18         return C;
19         if Q(C) ≥ T2 then
20             STOP all Find_C&CS instances;
```

2) Conversation Set Extraction Algorithm: The conversation set extraction is implemented in Algorithm 1. It takes as input an event sequence S and returns conversation sets, which are ordered with regard to their respective conversation set quality. It builds a first set C composed of one conversation equal to the first event of S. The events of S are then successively covered by recursively calling a new instance of $Find\_C\&CS(C, i)$. This procedure takes an event $e_i(\alpha_i)$ and tries to find, in the conversation set C, a conversation σ such that $e_i(\alpha_i)$ $correlates$ σ. If there exists such a conversation, the procedure builds for every possible correlation set (line 7) a new conversation set with the new conversation $\sigma.e_i(\alpha_i)$. Besides (line 13), an additional conversation set C_3 is built to consider that the event $e_i(\alpha_i)$ might also be the beginning of a new conversation. For every new conversation set that meets conversation invariants and quality, $Find\_C\&CS$ is recursively called (lines 12, 16).

If we consider the event sequence of Figure 1, Algorithm 1 extracts the conversation set of Figure 2, which holds 2 conversations whose events are correlated with the parameters in $\{id, trans, key, key2\}$.

Algorithm 1 may return several conversation sets ordered by quality if different correlations have been detected among successive events of S. In this case, the user has to choose the most appropriate conversation set with regard to its needs and knowledge.

B. Step2: Component Trace Extraction

The second step of our approach now generates as many trace sets as components found in the system SUL. This step, implemented by Algorithm 2, firstly covers the events of every conversation to identify the components of SUL (line 5). A the same time, it converts events by integrating the notions of input and output. In lines 6-8, every request or response is indeed doubled by separating the component source and destination. The component that executes the event is identified by a new assignment on the parameter idc injected to each event. Non-communicating events (neither requests or responses) are marked as outputs. Then, Algorithm 2 segments the resulting sequence σ' into subsequences, each capturing the behaviours of one component only of the set $Comp$ (lines 9, 10). The algorithm returns form every component $c \in Comp$ a set T_c that gathers the traces of the component c only.

Algorithm 2: Component Trace Extraction

```
   input : Conversation set C
   output: Trace sets T_{c_1} ... T_{c_n}
1  T := {};
2  foreach σ = e_1(α_1) ... e_k(α_k) ∈ C) do
3      σ' := ϵ;
4      foreach e_i(α_i) ∈ σ do
5          Comp := Comp ∪ {from(e_i(α_i)), to(e_i(α_i))};
6          σ' := σ'.!a_i({idc := from(e_i(α_i))} ∪ α_i);
7          if isReq(e_i(α_i)) ∨ isResp(e_i(α_i))) then
8              σ' := σ'.?e_i({idc := to(e_i(α_i))} ∪ α_i);
9      foreach c ∈ Comp do
10         T_c := T_c ∪ {σ' \ {e(α) ∈ σ' | (idc := c) ∉ α}}
```

C. Step 3: IOLTS Generation

Every trace set $T_c = \{\sigma_1, \ldots, \sigma_n\}$ is now lifted to the level of IOLTS.

A first IOLTS denoted L_c is obtained by transforming the traces of T_c to IOLTS paths. L_c is the IOLTS $L_c = \langle Q, q0, \Sigma, \rightarrow \rangle$ derived from T_c such that:

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

$$\frac{\sigma_i = a_1(\alpha_1)...a_k(\alpha_k)}{q0 \xrightarrow{e_1(\alpha_1)} (q_1 i, cl(\sigma_i))...(q_{k-1} i, Cl(\sigma_i)) \xrightarrow{a_k(\alpha_k)} q0}$$

By applying this trace set to IOLTS conversion on every component, we obtain the IOLTSs $L_{c_1}, \ldots, L_{c_n}$. Those IOLTSs are finally generalised by merging their equivalent states. The state merging is performed by means of the k-Tail algorithm [15], which is known to be a flexible state merging algorithm in the sense that it assembles the states sharing the same k-future, i.e. the same event sequences having the maximum length k.

Figure 2 depicts the IOLTSs generated from the conversation set given on top of the figure. Our approach has detected 6 components among the events of the two conversations. These have been converted to 6 trace sets, which capture the behaviours of each component. The traces sets have finally been converted to the IOLTSs depicted in the figure. In this example, we call k-Tail with $k := 2$.

With these IOLTSs, it becomes much easier to understand the general functioning of the whole system. In particular, it is now easier to understand that three services allow to interact with wholesalers. Two of them seem to be behavioural equivalent, but the last one is faulty.

IV. Related Work on Passive Model Learning

Passive mode learning includes techniques that passively recover models from a given set of samples, e.g., a set of execution traces. These are said passive as there is no direct interaction with the system under learning. Models are often generated by encoding sample sets with state diagrams whose equivalent states are merged. For instance, k-Tail has been later enhanced with Gk-tail to generate Extended Finite State Machines encoding data constraints [16]. Other approaches also enhance k-Tail to build more precise models [17], [18], [19]. kBehavior [20] is another kind of approach that generates models from a set of traces by taking every trace one after the other and by completing a finite-state automaton in such a way that it now accepts the trace. These previous passive algorithms usually yield big models, which may quickly become unreadable.

Some passive approaches dedicated to communicating systems have also been proposed. Mariani et al. proposed in [20] an automatic detection of failures in log files by means of model learning. This work extends kBehavior to support events combined with data. It segments an event log with two strategies: per component or per user. The former, which can be used with communicating systems, generates one model for each component. CSight [6] is another tool specialised in the model learning of communicating systems, where components exchange messages through synchronous channels. It is assumed that both the channels and components are known. Besides, CSight requires specific trace sets, which are segmented with one subset by component. CSight follows five stages: 1) log parsing and mining of invariants 2) generation of a concrete Finite State Machine (FSM) that captures the functioning of the whole system by recomposing the traces of the components; 3) generation of a more concise abstract FSM; 4) model refinement with invariants that must hold in FSMs, and 5) generation of Communicating FSM.

We have proposed in [21] a passive model learning algorithm for recovering models of component-based systems. The requirements considered in this approach are different from those of the above approaches. The main difference lies in the fact that the communications among components are assumed hidden (not available in event logs). The algorithm is hence specific to this assumption. Then, we have proposed the approach CkTail in [7] to generate models of communicating systems from event logs. Compared to CSight, we do not assume that the trace sets are already prepared. The novelty proposed by CkTail lies in its capability of detecting sessions in event logs. Compared to this work, we assume with CkTail that the components follow a strict behaviour: they cannot run multiple instances; requests are processed by a component on a first-come, first served basis. Besides, components follow the request–response exchange pattern

We showed that CkTail builds more precise models than the other approaches by better recognising sessions, but we also concluded that its requirements are too restrictive to be widely used. The approach proposed in the paper relaxes these assumptions and now supports any kind of communicating system.

V. Conclusion

This paper has proposed the design of an approach specialised into the recovery of formal models from event logs generated by communicating systems made up of concurrent components. The approach firstly explores the conversation set space that can be derived from an event log and is guided toward the most relevant conversation sets by means of invariants and conversation quality metrics. The latter can be adapted to define user preferences or system contexts. Then, the approach generates one trace set for every component along with one IOLTS expressing its behaviours. These IOLTS can be later used as documentation or for automatics analyses.

There are several issues which require further investigation before evaluating our approach. One of them is to be able to propose a good balance between model size, readability and precision. For instance, the generated IOLTSs may be very large on account of similar event sequences having different parameter values. We hence intend to add further

steps to raise the IOLTS abstraction level, while preserving the possibility to analyse of use concrete parameter values.

ACKNOWLEDGEMENT

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

REFERENCES

[1] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Leaps: Learning-based proactive security auditing for clouds," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 265–285.

[2] S. Salva and E. Blot, "Verifying the application of security measures in iot software systems with model learning," in *Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020*, M. van Sinderen, H. Fill, and L. A. Maciaszek, Eds. ScitePress, 2020, pp. 350–360.

[3] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 807–817.

[4] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, Nov 2008, pp. 117–126.

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

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568246

[7] S. Salva and E. Blot, "Cktail: Model learning of communicating systems," in *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020*, R. Ali, H. Kaindl, and L. A. Maciaszek, Eds. SCITEPRESS, 2020, pp. 27–38.

[8] A. Barros, G. Decker, M. Dumas, and F. Weber, "Correlation patterns in service-oriented architectures," in *Fundamental Approaches to Software Engineering*, M. B. Dwyer and A. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 245–259.

[9] OASIS Consortium, "Ws-bpel version 2.0," April 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

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

[11] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, Nov 2012.

[12] R. Vaarandi and M. Pihelgas, "Logcluster - a data clustering and pattern mining algorithm for event logs," in *2015 11th International Conference on Network and Service Management (CNSM)*, Nov 2015, pp. 1–7.

[13] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 167–177. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196340

[14] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," *CoRR*, vol. abs/1811.03509, 2018. [Online]. Available: http://arxiv.org/abs/1811.03509

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

[16] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE'08. New York, NY, USA: ACM, 2008, pp. 501–510.

[17] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 267–277.

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

[19] F. Pastore, D. Micucci, and L. Mariani, "Timed k-tail: Automatic inference of timed automata," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 401–411.

[20] L. Mariani and M. Pezze, "Dynamic detection of cots component incompatibility," *IEEE Software*, vol. 24, no. 5, pp. 76–85, 2007.

[21] S. Salva and E. Blot, "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.*, 2018, pp. 298–305.