Reverse Engineering Behavioural Models of IoT Devices

Sébastien Salva, Elliott Blot

Reverse Engineering Behavioural Models of IoT Devices

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

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

***Abstract*—This paper addresses the problem of recovering behavioural models from IoT devices in order to help engineers understand how they are functioning and audit them. We present a model learning approach called ASSESS, which takes as inputs execution traces collected from IoT devices and generates models called systems of Labelled Transition Systems (LTSs). ASSESS generates as many LTSs as components integrated and identified into a device. The approach is specialised to IoT devices as it takes into account two architectures often used to integrate components with this kind of system (cyclic functioning, loosely-coupled or decoupled architectures). We experimented the approach on two IoT devices and an IoT gateway to evaluate the model conciseness and the approach efficiency.**

***Keywords*-Reverse engineering; IoT; Model learning; Passive learning.**

I. Introduction

Internet connected devices, and especially Internet of Things (IoT), belong to the digital transformation trends proposed by industrial experts or advisory firms for several years. The IoT, which can be defined as a network of smart embedded devices connected to the Internet, is indeed a broad-based concept, transforming several uses from consumer devices to large-scale manufacturing. But, many customers and companies prefer staying away from the IoT hype because of the issues related to privacy and more generally to security. It is indeed manifest that IoT devices have to be audited before using them, in particular in the industry or in healthcare. Many companies chose to outsource the IoT development for saving costs, hence the IoT audit is rather done after the development. It is often carried out from the source code or from devices seen as black boxes. A common solution to audit such devices is to apply a reverse engineering process, which is usually done by hands. From a black-box, this process is required to understand how devices are functioning. Besides, it helps document the behaviours of IoT devices or IoT networks, and may serve to detect bugs or security issues.

In the literature, some papers dealing with the reverse engineering of IoT devices have been published recently [1], [2]. These approaches recover critical information or detect privacy issues from source codes, firmwares or chips. This paper proposes another approach called ASSESS (AnalySiS, Extraction, Separation, Synchronisation) to recover behavioural models from IoT devices. Our approach, which is based on the model learning concept, takes execution traces collected from IoT devices and generates models called systems of LTSs (Labelled Transition System). Model learning approaches [3], [4], [5], [6], [7], [8], [9] have proven to be valuable for retro-engineering models that can be exploited in several software engineering steps. Our approach advances the state of the art in these two points.

- It is specialised to IoT devices in the sense that their general functioning is taken into consideration while the model generation. We define an IoT device as an embedded device integrating several components and running in a cyclic way [10]. Several works focused on the component architectures of embedded devices, i.e. on how to compose them efficiently. It is often advised to use a loosely-coupled architecture [11], where components remain autonomous and allow middle-ware software to manage internal communication between them. With this kind of architecture, components are synchronised together. But, we also observed that IoT devices may also have a decoupled architecture, where the components operate independently. We consider both architectures for the model generation.
- Most of the model learning algorithms build one big model for a given system. Such models may quickly become uninterpretable. Instead, our approach builds as many LTSs as components detected in messages or logs collected from an IoT device. From these messages, our approach is able to build traces and infer systems of LTSs. Two strategies, which refer to the previous IoT device architectures, are proposed to synchronise LTSs together to form a complete model.

We have implemented a prototype tool to experiment our algorithms and appraise their benefits. We provide a preliminary evaluation in the paper made on two IoT devices. Besides, this experimentation also shows that ASSESS may be applied on an IoT gateway to recover a model expressing the behaviours of an IoT network, i.e. of the devices communicating with this gateway.

The paper is organized as follows: we recall some definitions about the LTS model in Section II. Our approach is presented in Section III. The next section shows some results of our experimentation. Section V summarises our contributions and draws some perspectives for future work.

II. The LTS model

We express the behaviours of components with Labelled Transition Systems (LTS). This 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 (silent) 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; $q0$ is the initial state;*
- *$\Sigma \cup \{\tau\} \subseteq \mathcal{L}_\tau$ is the finite set of actions, with τ the internal (unobservable) action;*
- *$\rightarrow \subseteq Q \times \Sigma \cup \{\tau\} \times Q$ is a finite set of transitions. A transition (q, a, q') is also denoted $q \xrightarrow{a} q'$.*

We use the generalised transition relation $\rightarrow$ to represent LTS paths: $q \xrightarrow{a_1 \dots a_n} q' =_{def} \exists q0 \dots q_n, q = q0 \xrightarrow{a_1} q_1 \dots q_{n-1} \xrightarrow{a_n} q_n = q'$. We also use the following notations on action sequences. The concatenation of two action sequences seq_1, $seq_2 \in \mathcal{L}_\tau^*$ is denoted $seq_1.seq_2$. ϵ denotes the empty sequence. A trace is a finite sequence of observable actions in $\mathcal{L}^*$.

To better match the functioning of IoT devices, we assume that an action has the form $a(\alpha)$ with a a label and α an assignment of parameters in P, with P the set of parameter assignments. For example, $switch(id := 115, cmd := on)$ is made up of the label "switch" followed by the assignment $(id := 115, cmd := on)$ of two parameters.

The use of LTSs allows to exploit the definitions related to the LTS composition. The integration of two components C_1 and C_2, modelled with LTSs, is often defined by two operations in the literature. 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* (the rest must happen independently). 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 a set of actions. We refer to [12] 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 2 (System of LTSs) *A system of LTSs SC is the couple $\langle S, C \rangle$ with $C = \{C_1, \dots, C_n\}$ a non empty set of LTSs, and S a set of synchronisation actions.*

III. The ASSESS Approach

This section presents our model learning approach, which aims at inferring system of LTSs from messages given by an IoT device. The later is seen as a black box and integrates components by means of a loosely coupled or a decoupled architecture. We assume that the components produce messages or logs which include component identifiers, i.e. parameter assignments allowing to identify components. But we consider that the component calls are hidden. This is usually the case with IoT devices integrating several sensors. Furthermore, the messages have to include timestamps for ordering them.

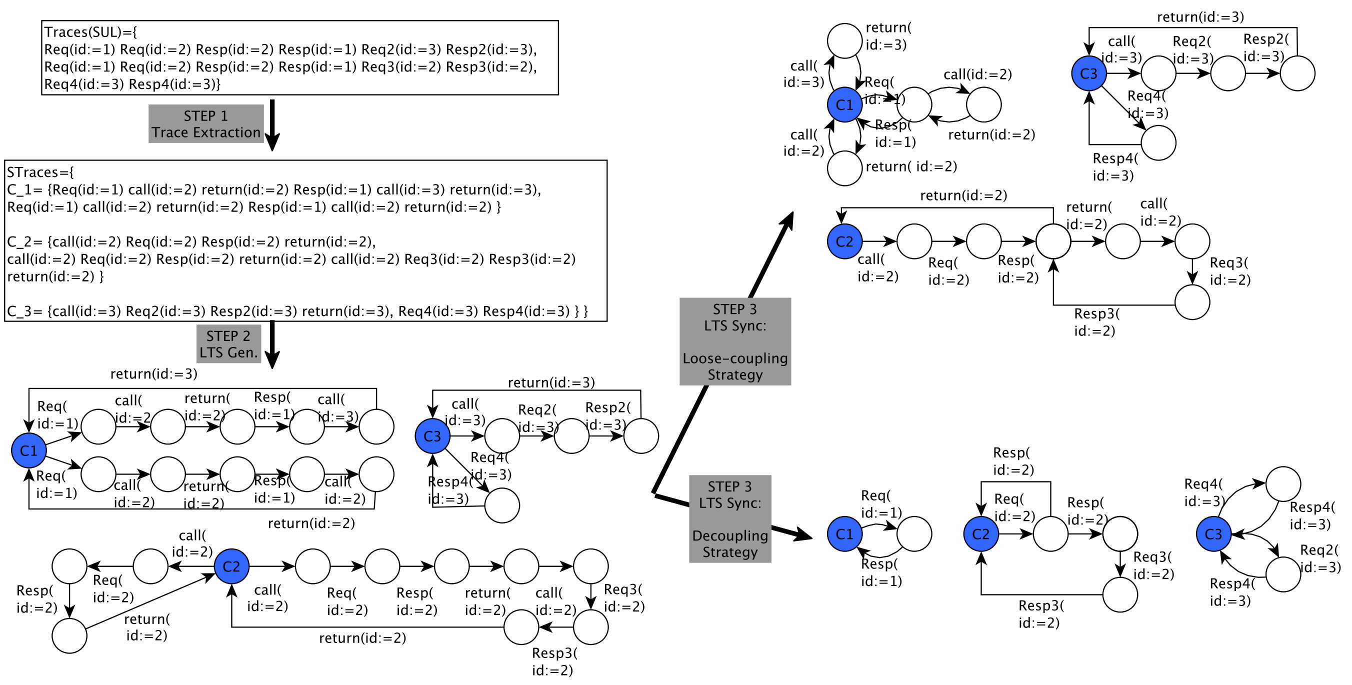
The list of messages is initially translated into a set of execution traces with our tool TFormat[1]. This one starts by filtering and formatting raw messages into actions by means of regular expressions. Then, the tool analyses the timestamps of every pair of successive actions and computes means of time intervals. It searches for gaps between actions (distinctive longer durations), which are usually observed when an execution trace ends and another one begins. The time gap detection is used for the trace extraction. We denote the trace set $Traces(\text{SUL})$ and assume that a trace has the form $a_1(\alpha_1)...a_k(\alpha_k)$.

The model generation is performed by three steps called "Trace Extraction", "LTS Generation", and "LTS Synchronisation". The last step proposes two LTS generation strategies called "LTS Loose-coupling" and "LTS Decoupling". These steps are illustrated with the example of Figure 1. In the first step, the traces of $Traces(\text{SUL})$ are analysed to detect component calls by covering the component identifiers found in actions. The example of Figure 1 lists 3 traces that capture the behaviours of two components (id:=1, id:=3), which call other components. The component calls are here detected whenever a new identifier is found (id:=2, id:=3). In a trace, the action sequences having different identifiers are extracted and replaced by synchronisation actions of the form $call(id)$ and $return(id)$ to express component calls, with id an identifier referring to a component. Next, the resulting traces are partitioned to gather the traces having the same identifier. We obtain 3 trace sets in our example of Figure 1.

The step "LTS Generation" transforms each previous trace set into a LTS. In this step, we take into account the general functioning of the IoT devices, which are usually designed to perform actions in a cyclic way. The traces are hence transformed into cyclic LTS paths, the later being joined on an initial state. Once every trace set is transformed into a LTS, we obtain a first system of LTSs $SC = \langle S, C \rangle$ with C the set of LTSs and S the set of synchronisation actions.

The last step transforms this system of LTSs to produce more general models with respect to the nature of the IoT devices. As stated earlier, we consider that these devices may integrate loosely-coupled or decoupled components. The strategy "LTS Loose-coupling" builds a system of LTSs $SC1$ such that $SC1$ allows repetitive calls of components, which are synchronised together. This is materialised by

[1]https://github.com/sasa27/TFormat


Traces(SUL)={
Req(id:=1) Req(id:=2) Resp(id:=2) Resp(id:=1) Req2(id:=3) Resp2(id:=3),
Req(id:=1) Req(id:=2) Resp(id:=2) Resp(id:=1) Req3(id:=2) Resp3(id:=2),
Req4(id:=3) Resp4(id:=3)}

STraces={
C_1= {Req(id:=1) call(id:=2) return(id:=2) Resp(id:=1) call(id:=3) return(id:=3),
Req(id:=1) call(id:=2) return(id:=2) Resp(id:=1) call(id:=2) return(id:=2) }

C_2= {call(id:=2) Req(id:=2) Resp(id:=2) return(id:=2),
call(id:=2) Req(id:=2) Resp(id:=2) return(id:=2) call(id:=2) Req3(id:=2) Resp3(id:=2)
return(id:=2) }

C_3= {call(id:=3) Req2(id:=3) Resp2(id:=3) return(id:=3), Req4(id:=3) Resp4(id:=3) } }

Figure 1: The ASSESS approach overview

replacing the sequences $q_1 \xrightarrow{call(id)\ return(id)} q_2$ with loops. Then, we apply the kTail algorithm [4]. kTail is a well-known approach that merges the (equivalent) states having the same k-future, i.e. the same event sequences having the maximum length k.

We obtain three LTSs in Figure 1(right-top side) expressing components that call each other. The strategy "LTS Decoupling" produces another system of LTSs $SC2$ from SC to express the behaviours of independent components. The synchronised actions are removed from the LTSs of SC. Then, the kTail algorithm is applied. We obtain three LTSs expressing autonomous components. Now, we detail these steps below.

A. Step 1: Trace Extraction

This step covers the traces of $Traces(\text{SUL})$ and the identifiers included in actions to detect implicit component calls and to gather the traces related to each component in separate trace sets. The following definition formalises the notion of component identification:

Definition 3 (Component identification) *Let $a_1(\alpha_1)$ be an action of $\mathcal{L}$. The component identifier of $a_1(\alpha_1)$ is given by the mapping $ID : \mathcal{L} \to P$, which gives the parameter assignment α' found in α_1 that identifies the component producing the action $a_1(\alpha_1)$.*
The component identifier of a sequence $a_1(\alpha_1)a_2(\alpha_2) \dots a_k(\alpha_k)$ is given by the mapping $ID_s : \mathcal{L}^* \to P$. $ID_s(a_1(\alpha_1)a_2(\alpha_2) \dots a_k(\alpha_k)) =_{def}$

$$\begin{cases} \alpha' \text{ iff } \forall a_i \notin \{call, return\} : ID(a_i(\alpha_i)) = \alpha'(1 \le i < k) \\ \{\} \text{ otherwise.} \end{cases}$$

For simplicity, we denote the mapping ID_s by ID in the remainder of the paper.

Algorithm 1: Component Trace Detection

```
input : $Traces$(SUL)
output: $STraces$
1 $Traces := \{\}$;
2 foreach $t = a_1(\alpha_1)a_2(\alpha_2)\dots a_k(\alpha_k) \in Traces(\text{SUL})$ do
3     $id := ID(a_1(\alpha_1))$; $T := \{\}$;
4     $T$:=Extract($t, T, id$);
5     $Traces := Traces \cup T$;
6 $STraces$:=GroupById($Traces$);
7 return $STraces$;
```

The Trace Extraction step is implemented with Algorithm 1, and its two procedures $Extract$ and $GroupById$. The algorithm covers every trace t of $Traces(\text{SUL})$, extracts the identifier id of the first running component found in the first action of t and calls the procedure $Extract$. The latter takes t, id and a set T used to store new traces. $Extract$ potentially splits t into several traces, each having one non empty component identifier. Then, the procedure $GroupById$ partitions all the traces given by $Extract$ and returns the set $STraces = \{C_1, C_2, \dots, C_n\}$ such that the traces of a set C_i exhibit the behaviour of one component only.

The procedure $Extract(t = a_1(\alpha_1)a_2(\alpha_2) \dots a_k(\alpha_k), T, id)$ is given in Algorithm 2. It covers the component identifiers in the actions of t to detect component calls.While covering the actions of t, if an identifier n different from $newid$ (first identifier of the current trace) is found (line 6), we assume that a new component has been called by the current one. In this case, the procedure searches for the sequence $a_{i+1}(\alpha_{i+1})...a_{j-1}(\alpha_{j-1})$ composed of actions having identifiers different from $newid$. This sequence is extracted and replaced by the synchronisation actions $call(n).return(n)$, which model the call of a component C_n. If the extracted sequence has more than one action, the

Algorithm 2: Procedure Extract

```
1  Procedure Extract($t = a_1(\alpha_1)a_2(\alpha_2)\dots a_k(\alpha_k), T, id$): $T$ is
2      $newid := Identifier(a_1(\alpha_1))$;
3      $t' := a_1(\alpha_1)$; $a_{k+1}(\alpha_{k+1}) = \epsilon$; $i := 1$;
4      while $i < k$ do
5          $n := ID(a_{i+1}(\alpha_{i+1}))$;
6          if $n == newid$ then
7              $t' := t'.a_{i+1}(\alpha_{i+1})$;
8              $j := i + 1$;
9          else
10             find smallest $j > i$ such that $ID(a_j(\alpha_j)) == newid$
               or $j := k + 1$;
11             $t' := t'.call(n)return(n).a_j(\alpha_j)$;
12             if $(j - i) > 2$ then
13                 Extract($a_{i+1}(\alpha_{i+1})\dots a_{j-1}(\alpha_{j-1}), T, id$);
14             else
15                 $t_n := call(n).a_{i+1}(\alpha_{i+1}).return(n)$;
16                 if $\exists t_2 \in T : ID(t_2) == n$ then
17                     $t_n := t_2.t_n$; $T := T \setminus \{t_2\}$;
18                 $T := T \cup \{t_n\}$;
19         $i := j$;
20     if $newid \neq id$ then
21         $t' := call(newid).t'.return(newid)$;
22     if $\exists t_2 \in T : ID(t_2) == newid$ then
23         $t' := t_2.t'$; $T := T \setminus \{t_2\}$;
24     $T := T \cup \{t'\}$;
25     return $T$;
```

procedure $Extract$ is recursively called (line 13). Otherwise, it builds a trace t_n composed of the action $a_{i+1}(\alpha_{i+1})$ surrounded by synchronisation actions. If there exists a trace t_2 in T having the identifier n, t_n is concatenated to t_2. t_n is added to the trace set T. Once the trace t is covered, we obtain a new trace t' including synchronisation actions. The procedure $Extract$ eventually checks whether t' has to be completed to express that this trace was produced by a component called by another one: if the identifier of t' is different from the identifier id given as input (line 20) then the trace t' is surrounded with $call(idnew)$ and $return(idnew)$. Finally, if there exists a trace t_2 in T having the component identifier $idnew$, then t' is concatenated to t_2. The final trace t' is added to T.

The procedure $GroupById(Traces) : STraces$, partitions the trace set $Traces$ in such a way that every subset holds traces sharing the same non empty component identifier. We partition $Trace$ by defining the trace equivalence relation $\sim_{id}$ and by extracting the equivalences classes of $Trace$ for $\sim_{id}$. Let $\sim_{id}$ on $\mathcal{L}^*$ be given by $\forall seq_1, seq_2 \in \mathcal{L}^*$, $seq_1 \sim_{id} seq_2$ iff $ID(seq_1) = ID(seq_2)$. The procedure $GroupById$ returns the partition $STraces = Trace/ \sim_{id}$.

B. Step 2: LTS Generation

At this stage, $STraces$ gathers n subsets with n the number of component identifiers found in the traces of $Traces(\mathrm{SUL})$. These subsets of traces are now transformed into LTSs. Intuitively, given T_1 in $STraces$, a trace of T_1 is lifted to the level of a LTS cyclic path. The LTS is obtained after joining the paths by means of a disjoint union on the state $q0$:

Definition 4 (LTS inference) *Let $T_1 \in STraces$ be a trace set. The LTS C_1 expressing the behaviours found in T_1 is the tuple $\langle Q, q0, \Sigma, \rightarrow\rangle$ where $q0$ is the initial state, and $Q, \Sigma, \rightarrow$ are defined by the following rule:*

$$\frac{t=a_1(\alpha_1)\dots a_k(\alpha_k), id=ID(t)}{q0 \xrightarrow{a_1(\alpha_1)} q_{id1}\dots q_{idk-1} \xrightarrow{a_k(\alpha_k)} q_0}$$

Once the LTS generation is completed, we obtain a first system of LTSs $SC = \langle S, C\rangle$ with C the set of LTSs derived from $STraces$ and S the set of synchronized actions.

C. Step 3: LTS Synchronization

Algorithm 3: LTS Synchronisation Strategies

```
1  Procedure Loose-coupling($SC = \langle S, \{C_1, C_2, \dots, C_n\}\rangle) : SC_1$ is
2      foreach $C_i = \langle Q, q0, \Sigma, \rightarrow\rangle \in C$ do
3          foreach $q_1 \xrightarrow{call(\sigma)return(\sigma)} q_2$ do
4              merge $q_1$ and $q_2$;
5          $C_i' := kTail(k = 2, C_i)$;
6      return $\langle S, \{C_1', C_2', \dots, C_n'\}\rangle$
7  Procedure Decoupling($SC = \langle S, \{C_1, C_2, \dots, C_n\}\rangle) : SC_2$ is
8      foreach $C_i = \langle Q, q0, \Sigma, \rightarrow\rangle \in C$ do
9          $C_i :=$ hide $S$ in $C_i$;
10         $C_i := \tau$-reduce $C_i$;
11         $C_i' := kTail(k = 2, C_i)$;
12     return $\langle S, \{C_1', C_2', \dots, C_n'\}\rangle$
```

This last step proposes two strategies to synchronise the LTSs of SC with regard to the architecture considered to integrate components together. Both strategies are implemented in Algorithm 3 with two procedures.

The strategy "LTS Loose-coupling" builds a new system of LTSs SC_1 from SC and keeps the transitions carrying synchronised actions. This strategy allows repetitive calls of components but also makes these calls optional by replacing the transition sequences of the form $q \xrightarrow{call(\sigma)return(\sigma)} q'$ by loops (lines 3,4).

The strategy "LTS Decoupling" gives another system of LTSs $SC2$ from SC by firstly hiding the synchronisation actions. The operator hide S in C_i transforms the transitions of C_i by replacing the actions of S with the non observable action τ. We then reduce C_i by removing the transition labelled by τ. Several algorithms are proposed in the literature to perform this LTS reduction with respect to a given LTS equivalence relation. But, as the LTSs generated by Step 2 have a simple structure (only one outgoing transition per state), we propose a lightweight LTS reduction operation denoted τ-reduction:

Definition 5 (τ-reduction) *Let $C_1 = \langle Q_1, q0_1, \Sigma, \rightarrow_1$ be a LTS. τ-reduction $C_1 =_{def} \langle Q_2, q0_2, \Sigma, \rightarrow_2\rangle$ where $Q_2, q0_2, \rightarrow_2$ are the minimal sets satisfying the following inference rules:*

$$\frac{q_1 \xrightarrow{a(\alpha)} q_2}{q1 \xrightarrow{a(\alpha)}_2 q_2} \qquad \frac{q_1 \xrightarrow{a(\alpha)} q_2 \xrightarrow{\tau\ldots\tau} q_3}{q1 \xrightarrow{a(\alpha)}_2 (q_2 q_3)} \qquad \frac{q_1 \xrightarrow{\tau\ldots\tau} q_2 \xrightarrow{a(\alpha)} q_3}{(q1 q_2) \xrightarrow{a_(\alpha)}_2 q_2}$$

kTail is finally applied on the LTSs achieved by both strategies. We use $k = 2$ as recommended in [7].

Both systems of LTSs SC_1 and SC_2 offer different points of view. With SC_1, the component calls are explicitly given, which offers the possibility of extracting a dependency graph of components showing how the components are hierarchically organised. With the system of LTSs SC_2, as the transitions carrying synchronised actions are removed, the parallel composition of the LTSs expresses the behaviours of asynchronous and autonomous components, which hence produce actions independently of the others. As it is illustrated in Figure 1, the second strategy returns more compact and general models.

IV. Preliminary Evaluation

We have implemented our approach in a tool, with which we began a first evaluation to answer to these two questions:

- RQ$_1$: can ASSESS extract more concise and readable models than the ones generated by kTail?
- RQ$_2$: how long does ASSESS take to generate models?

Setup: we applied ASSESS on two IoT devices and one IoT gateway. The first device (*exp.1*) is a smart thermostat controlling heat-pumps via infra-red, composed of 4 components (a Web server, two sensors, and a component that manages the heating mode). The second device (*exp.2*) is a Wifi IP camera that integrates 5 components. The IoT gateway (*exp.3*) was interconnected to 8 autonomous devices, which we consider as components for the experimentation. We collected HTTP messages from these systems and formatted them with our tool TFormat. The results and the tool are available here[2].

A. Question RQ$_1$

Procedure: we collected traces for every setup and ran ASSESS with its two strategies. We also ran kTail on the same trace sets for comparison purposes. Then, we measured the sizes of the generated models. These are given in Table I. Furthermore, with large trace sets, model learning might return "spaghetti"-like models, containing an uninterpretable mess of transitions. We compared the generated models to deduce whether ASSESS can significantly help reduce this spaghetti model problem by inferring one model per component.

Exp.	kTail		Loosely-coupled		Loosely-coupled without *call* and *return*		Decoupled	
	#states	#trans	#states	#trans	#states	#trans	#states	#trans
exp.1	52	90	116	208	61	118	31	54
exp.2	92	186	172	346	80	193	36	76
exp.3	349	419	426	552	362	439	310	339

Table I: Size of the LTSs obtained with kTail and ASSESS.

[2]https://github.com/Elblot/ASSESS

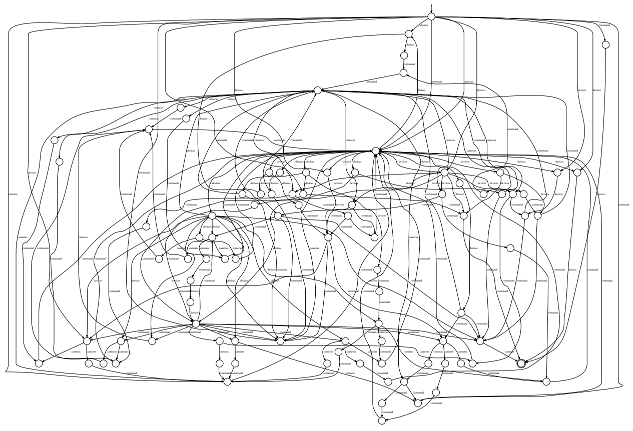


Figure 2: Overview of the models generated with kTail (*exp.3*)

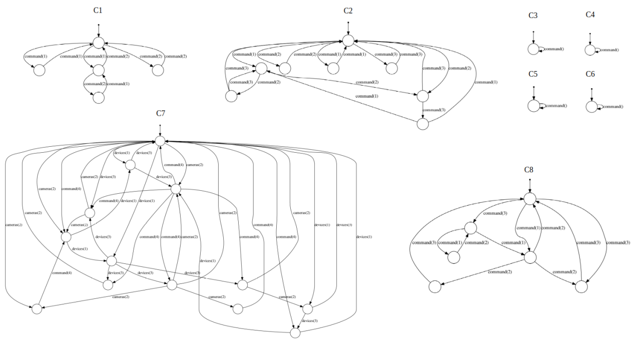


Figure 3: Overview of the models generated with ASSESS (*exp.3*)

Results: Table I shows that we obtain larger transition sets with the "LTS Loose-coupling" strategy. In average, the state number is increased by 77,33% in comparison to the results of kTail. This is due to the addition of transitions labelled by synchronisation actions, which show how components interact with one another. If we do not take into account these transitions, we obtain models whose sizes are close to the sizes of the models generated by kTail. With the "LTS Decoupling" strategy, we always obtain more concise models. The state number is reduced on average by 37,33% with this strategy. The state reduction is a consequence of the segmentation of the traces by our algorithm. We infer one LTS for each component, which is easier to reduce with kTail than one big model. Afterwards, we compared the models generated by kTail and ASSESS and manifestly concluded on these experimentations that the systems of LTSs are significantly more interpretable. Figure 2 shows an overview of the "spaghetti"-like model generated by kTail for *exp.3*. This model (even zoomed) is difficult to understand. Figure 3 illustrates the system of LTSs generated by ASSESS (second strategy). We believe that the later is more readable since every component is represented by its own model whose transition set is smaller. Besides, a system of LTSs sounds more adaptable to the user needs. For instance, an undesired component may be concealed to help focus on the others.

B. Question RQ$_2$

Procedure: to investigate RQ$_2$, we measured the execution times of ASSESS with several trace sets containing 10 to 35000 traces of around 150 events collected from *exp.3*. Experimentations were done on a computer with 1 Intel(R) CPU i5-6500 @ 3.2GHz and 16GB RAM. Figure 4 draws two curves showing the execution times measured with both strategies.

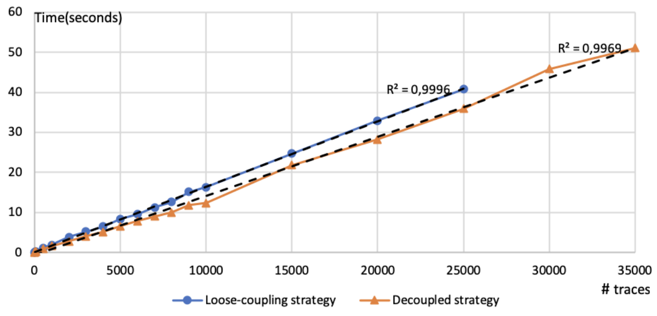


Figure 4: Executions times of ASSESS

Results: Figure 4 shows that ASSESS requires less than 60 seconds to builds models with the largest trace set. The tendency curves also confirm that the time complexity of both strategies is linear. Regarding the memory space complexity, we also observed a linear curve; we reached a memory limit between 25000 and 30000 traces (more than 3,5 millions of events) with the Loose-coupling strategy, and between 35000 and 40000 traces (more than 5 millions of events) with the Decoupled strategy. We hence believe that our tool can be used with real systems, even with a huge amount of messages.

V. Conclusion

The increase in IoT technology's popularity holds many benefits, but it is also accompanied by many concerns related to the IoT device reliability and security. Learning models from these devices may serve to audit them. But recovering models usable for inspection is still challenging. So far, most of the learning algorithms build big models and do not take into consideration the IoT device architectures. In this paper, we have presented ASSESS, a model learning method dedicated to IoT devices that recovers systems of LTSs. The method constructs execution traces from messages or logs, and generates LTSs that capture the behaviours of all the components of an IoT device and their synchronisations. Two strategies are proposed to adapt the model generation with regard to the loosely-coupled or decoupled architecture usually used to design embedded devices.

Our future work includes further evaluating ASSESS on other kinds of IoT devices, improving its effectiveness by devising parallel algorithms, and proposing other strategies to better match the available IoT architectures and frameworks.

References

[1] M. Tellez, S. El-Tawab, and M. H. Heydari, "Iot security attacks using reverse engineering methods on wsn applications," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, Dec 2016, pp. 182–187.

[2] O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, and Y. Oren, "Opening pandora's box: Effective techniques for reverse engineering iot devices," in *Smart Card Research and Advanced Applications*, T. Eisenbarth and Y. Teglia, Eds. Cham: Springer International Publishing, 2018, pp. 1–21.

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87 – 106, 1987.

[4] 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.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 213–224.

[6] K. Meinke and M. Sindhu, "Incremental learning-based testing for reactive systems," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, M. Gogolla and B. Wolff, Eds. Springer Berlin Heidelberg, 2011, vol. 6706, pp. 134–151.

[7] 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.

[8] 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.

[9] 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.

[10] L. Gomes and J. M. Fernandes, *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, 1st ed., ser. Premier Reference Source. Information Science Reference, 2009. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=CB7F775D3CF4C7A6E82A5331D620931E

[11] D. S. Stewart, "Designing software components for real-time applications," in *Proceedings of Embedded System Conference*, september 2000.

[12] M. van der Bijl, A. Rensink, and J. Tretmans, "Compositional testing with ioco," in *Formal Approaches to Software Testing*, A. Petrenko and A. Ulrich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 86–100.