Auditing the Security of IoT System With the Help of Model Learning and Model Checking

Elliott Blot
LIMOS, Université Clermont Auvergne, France
Email: elliott.blot@gmail.com

Warning / Attention

English

This thesis is an abandoned work. No more work from my side will ever be done on the subject, and it is not planned that I participate in the future works expressed in the perspectives. I just decided to finish the last 5% during my free time (Abstract/Acknowledgements, so expect a different tone on these two chapters). The thesis was not defended and did not provide me with any Ph.D. So yes, we can say it was useless work for me, and I lost 3 years on it :').

Français

Cette thèse est en état d'abandon. Aucun travail supplémentaire ne sera fourni de ma part sur les sujets de cette thèse ou sur aucun des sujets présentés dans les perspectives. J'ai juste décidé sur un coup de tête de terminer l'écriture des 5% restants pendant mon temps libre (Abstract/Acknowledgements, attendez-vous à un ton différent dans ces deux sections). Cette thèse n'a pas été soutenue et ne m'a pas permi d'avoir le titre de docteur. Donc oui on peut considérer que j'ai perdu 3 ans de ma vie là dessus :').

Acknowledgments

The first thanks will go to my lab colleagues, who have validated their thesis. It is always simpler to come to the office when some nice people help you to keep you smiling on such a rough adventure. And no hard feelings, everyone got chocoblasted at least once. I also want to thank friends from university, especially JN, for whom Peer reviewed some chapters of this thesis. Sorry, I inflicted you such pain for nothing at the end. The thesis was made in the loving memory of my Opel Astra Bertone, which left us during the 3 years of this PhD. I will never forget how you didn't break when there was snow just before a stop. A last thank you to Panzani because I eat a lot of pasta.

Résumé

Les objets connectés sont de plus en plus présents de nos jours, et tout et n'importe quoi peut avoir une version connectée. Parfois, ces objets sont absurdes, comme par exemple une fourchette (happyfork), mais il peut aussi s'agir d'objets d'une importance critique qui peuvent avoir un impact sur la sécurité physique de l'utilisateur, comme des objets médicaux, ou plus fréquemment une voiture. Cependant, ces objets sont souvent peu sécurisés, principalement parce que cela coûterait trop cher au constructeur, et parce que beaucoup n'y voient pas d'intérêt à les sécuriser (pourquoi quelqu'un voudrait pirater ma machine à café connectée ?). Pourtant, dans la réalité, pirater un objet aussi ridicule qu'un distributeur de papier toilette connécté (oui, ça existe : Rollscout) est très utile pour l'attaquant qui pourra ensuite s'en servir pour pouvoir attaquer une cible plus importante dans le même réseau ou sur Internet. L'exemple le plus parlant de ce genre de cas reste le malware Mirai, qui a été utilisé pour prendre le contrôle de milliers d'objets connectés pour les utiliser dans des attaques DDOS sur plusieurs sites Internet célèbres, les rendant inaccessibles pendant un certain temps. C'est pourquoi il est important de faire un audit de ces objets connectés, car ils peuvent représenter un point faible dans votre réseau et représenter une cible facile pour un attaquant qui voudrait les utiliser à votre insu. Dans cette thèse, nous proposons plusieurs méthodes basées sur les méthodes d'apprentissage de modèle passive, conçues spécialement pour les objets connectés ou les réseaux d'objets connectés, pour aider un auditeur durant cette analyse. Trois des méthodes presentées sont des méthodes d'apprentissage de modèle passive, qui permettent de générer des modèles représentant le comportement des objets connectés selon différents scénarios (COnfECt, ASSESS, CkTail). La dernière méthode présentée permet ensuite d'utiliser ces modèles produits pour analyser le comportement et vérifier si plusieurs mesures de sécurité sont intégrées par les objets connectés (SMProVer).

Abstract

Internet of Things (IoT) devices are more and more present today, and everything can have an IoT device version. Sometime, these devices are surprising, like a fork (happyfork), but they can also be critical devices that have an impact on the security of the user, like medical devices, or more frequently, a car. However, these devices are often poorly secured, mostly because it is too expensive for the constructor, and most people don't see any interest in securing them (Why someone will ever want to hack my smart coffee machine?). However, in reality, hacking a device as ridiculous as a toilet paper dispenser (yes, it exists: Rollscout) is still useful in order to process larger attacks on bigger targets in the same network or on the internet. The more known example is the malware Mirai, which was used to take control of thousands of IoT devices to transform them into botnets and perform DDOS on several famous websites, taking them down for a while. This is why it is necessary to perform an audit of IoT devices, as they can represent a weak point of your network or be an easy target for an attacker to create a botnet. In the thesis, we will propose several method based on passive model learning, designed especially for IoT devices or networks, in order to help an auditor during this analysis. Three methods proposed are passive model learning methods used to create a model representing the behavior of an IoT device in different scenarios (COnfECt, ASSESS, and CkTail). The last method presented will then use the model produced to analyze the behavior and check if several security measures are implemented in the devices (SMProVer).

Contents

Chapter 1

Introduction

1.1 Motivations

Internet of Things (IoT) can be defined as a network of smart embedded devices connected to the internet. More and more devices are connected to the internet, houses can be connected, cars can be connected, industries are connected. Such devices can even be used in sensitive domains, like in the healthcare, where several connected devices exist, that help peoples with diseases, and may control their health.

Connecting such devices gives some advantages to the user: easier collection of data, better traceability of the data, control with a smart phone, etc. However, it also provides some threats to the devices. As the devices are now connected, an attacker can try to remotely access them, causing a lot of damage to the devices and even endangering the life of the user in the case of devices used in sensitive domains. A searcher in medical device security showed that it is possible to hack remotely a pacemaker in order to send a discharge to an equipped patient [28]. Healthcare devices are not the only devices that can have a direct impact on the physical security of the user if they are compromised. In [40], two searchers succeeded in hacking a smart car driving on the highway and forcing it to stop. The two hackers could even remotely control the brakes, forcing their activation or preventing the conductor to do it. Even for less sensitive domains like smart houses or smart cities, the inconvenience of an attacker taking control of the system is non negligible and generally not wanted by the consumers that bought the IoT devices. Nobody wants that a stranger can control the lights or the heating system of its house, or can unlock it in the case of a smart lock.

Moreover, even if an attacker seems not to have interest in taking control of an IoT device that does not operate sensitive data, like, for example, a thermometer or a presence sensor, this device can still be used to attack a larger system. That is what was done with Mirai [7], a malware that infected several connected devices in 2016, then used them later to produce distributed denial of service (ddos) attacks on websites. Mirai is a malware that takes control of connected devices, turning

them into bots that can be used to perform attacks on bigger systems. Connected devices infected by Mirai constantly search for other connected devices around them. They then try to connect to them, using a list of default factory usernames and passwords, and install the malware on the devices where the connection was successful.

As demonstrated with Mirai, large-scale attacks can take advantage of the lack of security on existing connected devices, in the case of Mirai, the lack of security configuration forcing the user to redefine the default username and password. However, it is well known that such configuration should be implemented by connected devices [17]. It is then necessary to assess the devices provided to the public in order to be sure that these devices are secure. This process of analyzing a system in order to detect security flaws is called a security audit.

1.2 Goal of the Thesis

This work falls within the context of a security audit of IoT systems. A security audit of a system is the evaluation of the security of all the elements of the system. This security audit is not always done by the IoT device providers because this analysis can be long, difficult, and costly. That is why the goal of this thesis is to simplify this process of auditing for IoT systems by automating a part of this process.

To ease this audit and to automate it, it can be useful to represent the system in an abstract model. Such a model generalizes the system, allowing the auditor to have a better overview of the system and better understanding of the behavior of the system modeled. In this thesis, we present a method called SMProVer that helps an auditor determine if the components of a system, each represented by a model, implement security measures. IoT systems are generally composed of several components, for example, sensors, actuators, or gateways. A system is as weak as its weakest component; that is why we chose in this thesis to assess separately the different components of the system, because if a single component has a weak security, that is the whole system that has a weak security. This method aims to automate the audit, meaning that we also need that the models of the components assessed are automatically generated. In this thesis, several methods are proposed to produce the models of the components, called COnfECt, ASSESS, and CkTail, where each model represents a component of the system.

1.3 Contributions

The contributions of the thesis are the following :

- COnfECt and ASSESS, two methods that aim to generate the models representing the behavior of the components of a component-based system, in the case where the communications between the components are not visible.

These methods could be used, for example, for a complex embedded device composed of several sensors or actuators. The first one, COnfECt, is designed to produce models of any component-based system, while ASSESS is a specialization of COnfECt designed for IoT systems. We also provide an evaluation of the methods done with the help of their implementations, where the results obtained with both methods are compared and where their performances are shown.

- CkTail, a method that generates the models representing the behavior of the components of an IoT system in the case where the communications between the components are visible. This method goes further by also producing for each component a dependency graph representing how the component interacts with the others. This view can be helpful, for example, to determine which components an attacker has to attack in order to cause more damage to the system. This method can be used, for example, for an IoT network composed of several devices communicating the ones with the others. We also provide an evaluation, done with the help of our implementation, where the method is compared to other model learning methods in terms of precision of the models produced and performance. The precision of the dependency graphs produced is also evaluated.

- SMProVer is a method that helps an auditor audit an IoT system by verifying that some security measures are implemented by every component of the system. This method uses the models automatically produced by our method CkTail and verifies on them some properties that represent the security measures to verify with the help of formal methods. We also propose several properties, representing security measures found in the literature that can be verified on an IoT system to assess its security. Once again, the method was implemented, and the implementation was used to evaluate its precision and the performance.

1.4 Thesis Organisation

This thesis is structured as follows:

Chapter 2 discusses the existing works on automated audit methods and which parts of the methods still need automation. We also present some techniques used to automatically generate a model of a system, discuss which ones are the most adapted to IoT systems, and describe their drawbacks.

In Chapter 3, we present the two methods COnfECt and ASSESS, designed for generating models of the components of a component-based system, such as a complex IoT device composed of several sensors and actuators. Moreover, the method also produces for each component a dependency graph representing how the components interact with each other. The two methods are presented in the

same chapter because they are very similar, ASSESS being an adaptation of COnfECt specialized for IoT systems. Both methods are described one by one, and a mutual evaluation is provided where the methods are compared in terms of precision and performance.

In Chapter 4, we present the method CkTail, designed for generating models of a component-based system where the components communicate the ones with the others and where these communications are not hidden. Such systems are also called communicating systems. The method is presented in details, and its evaluation is also provided, where its precision and performance are compared with other methods designed for communicating systems found in the literature.

In Chapter 5, we present the method SMProVer, which aims to help an auditor verify that security measures are implemented by every component of the system. This method uses the models produced by CkTail and properties written by an expert representing the security measures, and uses formal methods to verify the properties on the models. The evaluation provided in this chapter aims to show the precision and the performance of the implementation of the method.

Finally, in Chapter 6, we conclude the thesis by summarizing the contributions and providing some perspectives and works in progress that complete the works presented in the previous chapters.

Chapter 2

Related Works

2.1 Security Analysis

A lot of works presented the main challenges in IoT security and threats or vulnerabilities that can be found in the devices. Several works are proposed to audit IoT systems in order to face these threats, like in [42, 30], where they propose to audit the system with the help of a threat model that is produced from the recommendations provided by databases like the Owasp [45] or the ENISA[17] and data collected from the system via external tools, and use them to assess the security of different aspects of the IoT device, using penetration testing tools or manually. Some other works are centered only on a specific aspect of the IoT system, like, for example, a specific protocol. In [58], the authors propose an approach allowing the owner of an IoT device to assess the TLS traffic sent by the device to the cloud owned by the manufacturer. The user has the possibility to decrypt old encrypted messages in order to verify that only data necessary for the functioning of the device was sent to the manufacturer.

Several other methods need a model of the system in order to assess its security. In [22], the authors use a model produced manually from data collected from the system and use it to find potential attack scenarios, evaluate security metrics, and assess the effectiveness of different defense strategies. Model-based testing methods are other types of methods that necessitate an abstract model of the system. In these methods, the tests can be produced manually from the model and then applied in the system to experiment with it [24, 1, 53, 38]. Other model-based testing methods monitor the system to produce an abstract model of the system in order to verify that the system satisfy some properties [51, 12, 35]. Several tools of model-based testing were designed for IoT systems, like, for example, the tool IoTSAT [41], where the behavior of the system is modeled with SMT logic and assessed with the help of SMT constraints representing the goals and the capacities of the attacker in order to identify threats, potential attacks, and the resistance of the system against them. In [51], they propose a tool that scans the environment in order to model it from the data collected. The model is then assessed to identify

privacy threats in the IoT environment.

However, each of these methods necessitates that the user produce manually a specific model of the system and the corresponding properties to verify. In this thesis, we will present a method that uses models generated automatically to verify a set of generic properties on them. In the literature, there exist methods that produce automatically an abstract model of the system, called model learning methods, that will be described in the next section.

2.2 Model Learning

In this thesis, the models of the IoT systems will be automatically generated. In order to do that, we will use model learning techniques that aim to infer a model of the system under learning, denoted SUL. A model learning method can be defined as *a set of methods that infer a specification by gathering and analyzing system executions and concisely summarizing the frequent interaction patterns as state machines that capture the system behavior* [5]. The models produced give a more general view of the system and can then be used as documentation, for test generation, or to detect bugs with the help of formal methods. In the literature, model learning methods can be separated into two groups, the active methods and the passive methods, that are described with more details in the following.

2.2.1 Active Methods

Active methods [6, 26, 25] consist of actively querying the system or the user in order to collect information that is then used to produce a model representing the behaviors of SUL. One of the main active methods is the $\mathcal{L}^*$ algorithm [6], which iteratively learns a model by querying an oracle. The algorithm sends a model that hypothetically represents SUL to the oracle that then validates the model or returns counterexamples to the algorithm that will be used to upgrade the hypothesised model until it finally gets one that represents the system. This algorithm has been ported to various systems and contexts [48, 4, 8, 26, 25], and has also been optimized to mostly reduce the query number [14, 27, 2]. However, having an oracle knowing all about the behaviors of SUL is a strong assumption.

An other solution of the active method is called incremental learning [15]. With incremental learning techniques, positive or negative samples, extracted from SUL, are successively received. Several models are incrementally built in such a way that if a new observation is not consistent with the current model, the latter is suitably modified. These methods require that the system is controllable, i.e., we can control the output of the system, as they need to query directly the system to collect the information.

Definition 1 *Controllability We say that a system is controllable if, for each output of the system, there exists an input that leads to this output. A system that is not controllable is called uncontrollable.*

Uncontrollable systems are difficult to query, making it difficult to use active methods on them. IoT systems are generally composed of sensors that are uncontrollable, as they periodically send data in the network without having received any orders as inputs.

That is why in this thesis, we choose to use a different type of model learning method, called passive model learning method, that uses data collected from SUL to produce the model that can be easily used on uncontrollable systems.

2.2.2 Passive Methods

The passive model learning methods aim to produce an abstract model of the system by taking as inputs data extracted from the system, like, for example, a log or execution traces. The main passive learning methods in the literature are kTail [11] and kbehavior [37].

kTail takes as input a trace set and produces a Finite-State Automaton (FSA). The FSA is produced in two steps. First, it builds a tree, whose edges are labeled with the events found in traces, called a Prefix Tree Acceptor. Then kTail transforms this tree into a FSA with the help of an algorithm that merges the equivalent states. Two states are equivalent if they have the same $kfuture$, i.e., the same futures of length k or less.

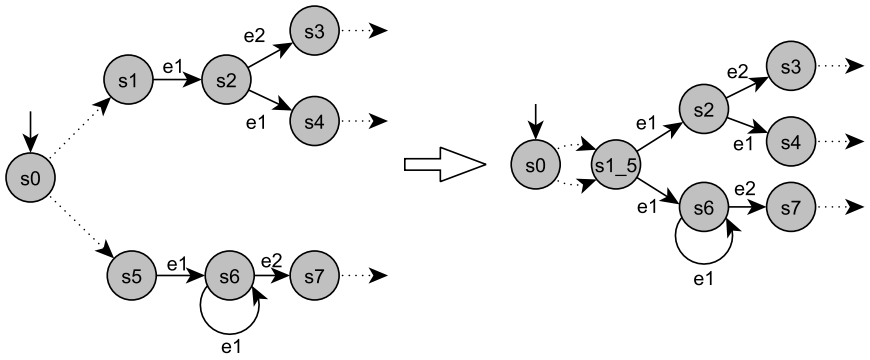


Figure 2.1: Example of states with the same $2futur$ that merge.

Figure 2.1 shows an example of states with the same $kfutur$, merged by this algorithm taking $k = 2$, also denoted $2future$. In this figure, the two states $s1$ and $s5$ have the same futures of length 2 or less: $\{e1e1, e1e2\}$. These two states are then merged into a new state called $s1\_5$ that cumulates the input and output transitions of both states $s1$ and $s5$. In the following of the thesis, the algorithm used in the last step of kTail that merges the states is denoted the kTail algorithm.

kTail was extended with Gk-tail [32], which produces Extended Finite State Machines (EFSM), that are FSMs encoding data constraints on the labels of the transitions. More recently, kTail was improved in order to produce timed automata

with Tk-tail [46]. The kTail algorithm was also used by several other methods like Perfume [44] or Synoptic [10] that generate temporal invariants from the log given as inputs that have to be satisfied by the model produced. The different methods presented in this thesis will also use the kTail algorithm for merging equivalent states in order to generalize and reduce the size of the models produced.

An other approach of passive model learning methods was designed by Mariani and his co-authors in [37], called kBehavior. kBehavior generates an FSA from a set of traces by taking every trace one after one and by completing the FSA in such a way that it now accepts the trace. More precisely, whenever a new trace is submitted to kBehavior, it first identifies the sub-traces that are accepted by a sub-automata in the current FSA (the sub-traces must have a minimal length k, otherwise they are considered too short to be relevant). Then, kBehavior extends the model with the addition of new branches that suitably connect the identified sub-automata, producing a new version of the model that accepts the entire trace.

The main flaw of the passive model learning method is the size of the model produced, which may be unreadable. In this thesis, we address this problem by splitting the model into several smaller and more readable models. Classical passive model learning methods produce only one model for the whole system. However, in the context of an IoT system, the system can be composed of several components, like sensors, actuators, or web services, that can depend on each other or be independent. Modeling each of these components separately could lead to smaller and more readable models. In this thesis, three passive model learning methods designed for IoT systems are presented: two of them are designed to produce the models of the components, where the communications between the components are hidden, and the last one is designed to produce the models of the components, where the communications between the components are shown in the log.

2.2.3 For Component Based Systems

Several model learning methods were designed for component-based systems, both active and passive.

Groz et al. proposed in [23] an active model learning method that produces a controllable approximation of the components via active testing. The models of the components are produced in isolation, and the components have to be testable and so controllable to be modeled. Moreover, the components have to be known in order to produce their models. A recent work proposes to test a system with unknown components by means of a SAT solving method [47]. In [54], Tappler and his co-authors propose a model-based testing technique for IoT systems based on the generation of models from multiple implementations of a common specification, which are later pair-wise cross-checked for equivalence. Any counterexample is flagged as suspicious and has to be assessed manually by the user. As stated in the previous section, active model learning methods cannot be applied to every IoT system, as they can be composed of uncontrollable components, like sensors. We

choose, in this thesis, to concentrate on passive model learning methods designed for component-based systems in order to take such uncontrollable components into account.

In the literature, two passive model learning methods were produced: CSight and the method provided by Mariani et al. in [36]. In the following, we refer to the last method with LFkbehavior. CSight [9] infers models of a communicating system where the components communicate via synchronous channels. The channels and the components have to be known and given to the method. CSight also requires that the trace set captures this notion of channels: the trace set is segmented into subsets, each representing a component. The traces are composed of input and output events. CSight has five main steps: 1) log parsing and mining of invariants that must hold in the models 2) generation of a concrete FSM that captures the functioning of the whole system by recomposing the traces of the different components 3) generation of a more concise abstract FSM 4) model refinement with invariants, and 5) generation of Communicating FSM (CFSM). The models produced show how the components interact with each other with the help of the channels and of the input/outputs. When a component sends an output, an other component receives an input with the same information. CSight needs that the communications between the components are shown in the traces; no interaction between the components will be found in the model if no communication between them is found. In this thesis, we will present passive model learning methods that can be applied in the case where the communications between the components are hidden and that can mine their interactions. An other flaw of CSight is that the invariant mining and satisfiability checking steps are costly and prevent the tool provided from taking as inputs medium to large trace sets. Moreover, the number of components has to be known by CSight, where the methods presented in this thesis do not need this information.

LFkbehavior, an other passive model learning method that can be used with component-based systems, was proposed in [36] as an automatic detection of failures by means of model learning. The event log is first segmented into sub-traces according to one of the following strategies: per component or per user. The method kBehavior is then used to produce for each sub-trace a model. The first strategy can be used with component-based systems, as it generates one model per component.

We observed that the passive model learning methods designed for component-based systems found in the literature lack precision due to the lack of session recognition in the log. We call a session a behavior of the system from its initial state to one of its final states. In this thesis, we will present a passive model learning method designed for component-based systems that aims to produce more precise models by detecting the session in the log.

Chapter 3

COnfECt and ASSESS: Model learning of components based systems

3.1 Introduction

Models are useful to analyze systems via model-based testing or formal verification. However, the difficulty in designing models is a strong barrier for the adoption of such methods in the industry. To help experts in the production of models, model learning methods were developed. I presented earlier that methods called passive model learning methods offer some advantages to learning models from communicating systems, as the system does not have to be resettable or testable and so controllable. Passive model learning methods take input data that was collected from the system, like, for example, execution traces. Classic passive model learning methods, like, for example, kTail [11] or kBehaviour [37], suffer from one main issue: the model obtained can be huge and difficult to understand. The system is indeed considered as a single block, and only one general model is produced. However, IoT systems are often composed of several components, like sensors or actuators, that are repeatedly queried. Modelling such components in separate models could lead to smaller models being easier to understand.

In this chapter, we present two passive model learning methods that aim to generate a system of labeled Transition Systems (LTSs) from a set of events, where each LTS represents the behavior of a component of the system and contains synchronization actions that show how the component interacts with the others. These two methods are designed to produce the model of a component-based system where the communications are unobservable, like, for example, a complex embedded device composed of several sensors and actuators. A third model learning method designed for communicating systems, where the components communicate the ones with the others, like, for example, a network of IoT devices, will be presented in the next chapter. The first method presented in this chapter, called

COnfECt (COrrelate Extract Compose), is a general method targeting any type of component-based system. The second one, called ASSESS (AnalySiS Extraction Separation Synchronization), is a specialization of COnfECt for IoT devices. These two methods take as inputs a log of raw events that can be collected with the help of a monitoring tool. They are composed of four main steps called *Trace Formatting*, *Trace Analysis & Extraction*, *LTS Generation*, and *LTS synchronization*. The first step extracts formatted traces from the raw events of the log; the second step analyzes these traces to extract sub-traces, each containing the behavior of only one component. The third step uses these sub-traces to produce the first models of the components, and finally the last step generalizes these models according to a strategy chosen by the user to produce a system of LTSs.

Both methods have been implemented and are available in github[1]. An evaluation is given in this chapter, comparing COnfECt and ASSESS with kTail on several points: component detection, readability of the model, rate of valid traces accepted by the model, rate of invalid traces accepted by the model, and scalability.

This chapter is structured as follows: first some preliminary definitions are given in Section 3.2, then an overview of the methods is given in Section 3.3. Afterwards, both methods are described, starting with COnfECt in Section 3.4 and followed by ASSESS in Section 3.5. An evaluation of the methods is provided in Section 3.6. Finally, Section 3.7 summarizes the chapter.

3.2 Preliminary

Both methods aim to produce models of component-based systems. The behavior of each component of the system is expressed in a labeled Transition System (LTS). An LTS is defined in terms of states and transitions labeled by actions, taken from a general set denoted $\mathcal{L}$, expressing what happens in the system. τ is a special symbol encoding an internal (unobservable) action. The set $\mathcal{L} \cup \tau$ is denoted $\mathcal{L}_\tau$.

Definition 2 (LTS) *A labeled 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 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* $q \xrightarrow{a} q'$.

To ease the readability, an LTS path $q_1 \dots q_{n-1}, q \xrightarrow{a_1} q_1 \dots q_{n-1} \xrightarrow{a_n} q'$ is denoted $q \xrightarrow{a_1 \dots a_n} q'$. The concatenation of two actions sequences $\sigma_1, \sigma_2 \in \mathcal{L}_\tau^*$ is denoted $\sigma_1.\sigma_2$. ϵ denotes the empty sequence.

[1] https://github.com/Elblot

We denote $C_1 \parallel C_2$, the parallel composition of two LTSs C_1 and C_2, which synchronizes their synchronization actions.

Definition 3 (LTSs composition) *Let $C_i = \langle Q_i, s0_i, \Sigma_i, \rightarrow_i \rangle_{i=1,2}$ be two LTSs, such that $\Sigma_1^I \cap \Sigma_2^I = \Sigma_1^O \cap \Sigma_2^O = \emptyset$. $C_1 \parallel C_2 = \langle Q, q0, \Sigma, \rightarrow \rangle$ such that:*

- $Q = \{q_1 \parallel q_2 \mid q_1 \in Q_1, q_2 \in Q_2\}$, $q0 = q0_1 \parallel q0_2$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$,
- $\rightarrow$ *is the minimal set satisfying the following inference rules:*
  - $q_1 \xrightarrow{a} q_1'$, $a \notin \Sigma_2 \vdash q_1 \parallel q_2 \xrightarrow{a} q_1' \parallel q_2$
  - $q_2 \xrightarrow{a} q_2'$, $a \notin \Sigma_1 \vdash q_1 \parallel q_2 \xrightarrow{a} q_1 \parallel q_2'$
  - $q_1 \xrightarrow{a} q_1', q_2 \xrightarrow{a} q_2', a \neq \tau \vdash q_1 \parallel q_2 \xrightarrow{a} q_1' \parallel q_2'$

It is then possible to hide the synchronization actions between C_1 and C_2 to keep only visible the communications with the environment within the models. We refer to [56] for the definitions of this operator. This principle of LTSs composition leads to a model called system of LTSs, which describes the complete component-based system:

Definition 4 (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 synchronization actions.*

3.3 Approaches Overview

The goal of COnfECt and ASSESS is to infer a system of LTSs SC, which captures the behaviors of the components of the system under learning (SUL) and their synchronizations. The method takes as input a log of raw events collected from SUL with the help of monitoring tools. SUL can be indeterministic, uncontrollable, have cycles among its internal states, and is considered a black box. However, SUL has to obey some restrictions:

- H_1**:** the communications among the components of SUL are not observable.
- H_2**:** Only one component can run at a time from its initial state to one of its final states. We consider that the events of the log are collected in a synchronous environment and include a timestamp to order them.

Moreover, each method relies on an additional specific assumption:

- $H_{COnfECt}$**:** There is a single first component called the root component that calls the others during its execution.

- H_{ASSESS}**:** The events contain an identifier representing the component that produced it.

With COnfECt, we suppose that the components follow a procedural behavior. There is a single first component C_1 that calls other components, called the root component. When C_1 calls another component C_2, C_1 waits that C_2 ends its execution. The called component C_2 runs from its initial state to one of its final states and can call other components during its execution. Once C_2 has ended its execution, C_1 continues its state execution where it paused when it called C_2. With ASSESS, we also suppose that the components follow this procedural behavior, but several root components are allowed.

We do not assume with COnfECt that components are identified. COnfECt proposes several strategies, one of them uses string metrics to determine if two events come from the same component. However, this strategy needs that the user defines thresholds for the metrics, but defining these thresholds is a difficult task. Another simpler strategy considers that the components are identified with an identifier. ASSESS only uses this last strategy.

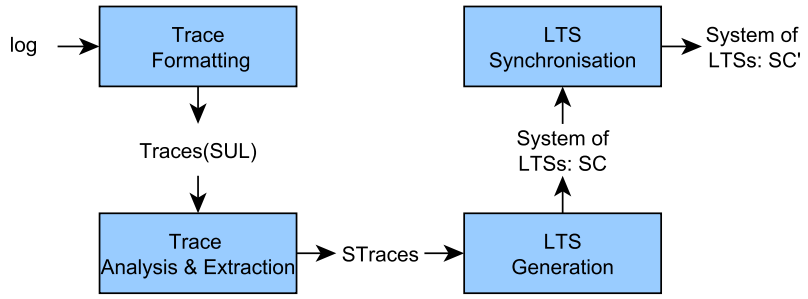


Figure 3.1: COnfECt and ASSESS overview.

COnfECt and ASSESS are composed of 4 steps, as shown in Figure 3.1. In the first step, *Trace Formatting*, the raw events extracted from SUL are transformed, and grouped in a set of formatted traces $Traces(SUL)$. In the second step *Trace Analysis & Extraction*, $Traces(SUL)$ is analysed and partitioned to form a new set $STraces$, where each trace set contains the behavior of only one component. In the third step *LTS Generation*, the traces of $STraces$ are used to produce the first system of LTSs SC, representing the components of the system. Finally, in the last step *LTS synchronization*, SC is transformed into a new system of LTSs SC', according to different synchronization strategies.

The first step *Trace Formatting* is identical for both methods, but there are major differences between the methods during the execution of the other steps. The step *Trace Analysis & Extraction* of ASSESS is designed to be simpler than the one of COnfECt by using component identifiers. The step *LTS Generation* of ASSESS is designed to generate models that fit better with IoT devices than the ones gener-

ated by COnfECt. Finally, the last step *LTS synchronization* of ASSESS proposes different synchronization strategies than the ones proposed by COnfECt. Each of these steps will be described with more details for both methods, starting with COnfECt in the following, and illustrated with an example of a system composed of three components.

3.4 Description of COnfECt

3.4.1 Trace Formatting

COnfECt takes as input a log of raw events, which are ordered by their timestamps. The goal of this step is to transform these raw events into formatted traces that can be easily processed by the methods. The events are transformed into actions of the form $a(\alpha)$, with a a label and α parameter assignments. For example, the action $swicth(idx := 115, cmd := on)$ is made up of the label "$switch$" followed by the assignment of two parameters. The events are transformed with the help of regular expressions that match the events and detect labels, parameters, and timestamps. These regular expressions can be built manually for small systems. However, for larger and more complex systems, these regular expressions can be derived from patterns mined from logs. Several works address this problem [36, 21, 34, 55, 39, 61] and propose approaches and tools to mine patterns. Figure 3.2 shows an example of raw events and a regular expression that can be applied. The example of formatted trace of the figure is built from the four raw events and contains four actions. The regular expression given in this example matches the first raw event of the figure and helps to produce the first action of the trace in the figure. We can see in the actions a parameter "idx" that will be used as the identifier of the components.

The set of actions is then split into traces according to different strategies: using component identifiers, using timestamps, or by combining the two previous strategies on the traces obtained. The first strategy, using component identifiers, consists of putting in the same trace the actions that have the same identifier. For example, with the raw events of Figure 3.2, by using this strategy we obtain 2 traces: one with the first and fourth actions, and one with the second and third actions. The second strategy consists of analyzing the time delay between consecutive actions. A gap between two actions represents a separation between two traces, as it potentially marks the end of an execution and the beginning of a new one.

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

3.4.2 Trace Analysis and Extraction

This step of COnfECt aims to analyze the traces of $Traces(SUL)$ and produce a new set $STraces$, which contains trace sets representing only one component. To

```
Example of raw events:
e1: Jul 18, 2018 08:52:26.696766000
    CET;Protocol=HTTP;Verb=GET;Uri=/devices?idx=25 HTTP/1.1;
e2: Jul 18, 2018 08:52:30.362482000
    CET;Protocol=HTTP;Verb=GET;Uri=/json.htm?type=command&
    param=udevice&idx=115&nvalue=0&svalue=15.00 HTTP/1.1;
e3: Jul 18, 2018 08:52:30.522163000
    CET;Protocol=HTTP;HTTP/1.1;status=200 response=OK idx=115;
e4: Jul 18, 2018 08:52:31.598645000
    CET;Protocol=HTTP;HTTP/1.1;status=200 response=OK idx=25
    data=<script...>;

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

Example of formatted trace:
e1: /devices(Protocol=HTTP;Verb=GET;Uri=/devices;idx=25)
e2: /json.htm(;Verb=GET
    Uri=/json.htm;type=command;param=udevice;idx=115;nvalue=0;
    svalue=15.00)
e3: OK(Protocol=HTTP;status=200;response=OK;idx=115)
e4: OK(Protocol=HTTP;status=200;response=OK;idx=25;
    data=<script...>)
```

Figure 3.2: Example of 4 raw events collected from a connected thermostat device. The regular expression retrieves a label and 4 parameters here. The label expression will be the label of the action in the formatted traces. This regular expression matches the first raw event.

do so, COnfECt covers the traces of $Traces(SUL)$, and uses a coefficient, defined by the user, to separate the actions that come from different components.

Algorithm 1: ConfECt 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** $t \in Traces(\text{SUL})$ **do**
4 $\quad \sigma'_1\sigma'_2 \ldots \sigma'_k$=Inspect($t$);
5 $\quad STraces$=Extract($\sigma'_1\sigma'_2 \ldots \sigma'_k, T_1$);
6 return $STraces$;

COnfECt proceeds as shown in Algorithm 1. For each trace of $Traces(SUL)$ it first splits them into sub-sequences with the procedure *Inspect*. Then, these sub-sequences are split with the help of the procedure *Extract*, which separates the

sub-sequences that come from different components. The procedure $Inspect$ necessitates that the user define a *Correlation coefficient*. This coefficient evaluates the correlation between actions in the traces of $Traces(SUL)$, representing the degree to which successive actions are related in the traces of $Traces(SUL)$. The *Correlation coefficient* between two actions is defined as a utility function, representing user preferences. These preferences are represented by using the technique *Simple Additive Weighting* [59].

Definition 5 (Correlation coefficient) *Let $a_1(\alpha_1)$, $a_2(\alpha_2) \in \mathcal{L}$ and $f_1, \dots f_k$ be correlation factors. $Corr(a_1(\alpha_1), a_2(\alpha_2))$ is a utility function, defined as:*
$0 \leq Corr(a_1(\alpha_1), a_2(\alpha_2)) = \sum_{i=1}^{k} f_i(a_1(\alpha_1), a_2(\alpha_2)).w_i \leq 1$ *with*
$0 \leq f_i(a_1(\alpha_1), a_2(\alpha_2)) \leq 1$, $w_i \in \mathbb{R}_0^+$ *and* $\sum_{i=1}^{k} w_i = 1$.

The correlation factors are specific to the context of the system and have to be written by an expert. The more precise the factors are, the more precise the separation of the actions of the traces will be. Some examples of correlation factors are given below:

- $f_1(a_1(\alpha_1), a_2(\alpha_1))$= 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 $\text{freq}(a_1a_2)$ the frequency of having the two labels a_1,a_2 one after the other in $Traces(\text{SUL})$ and $\text{freq}(a_1)$ the frequency of having the label a_1. This factor used in text mining computes the frequency of the term a_1a_2 in $Traces(\text{SUL})$ over a_1, and over a_2 to avoid the bias of getting a low factor when a_1 is greatly encountered (resp. a_2);

The first correlation factor assumes that every action includes an identifier of the component that produced it. This factor has a high precision but is specific to some systems, as not every system uses identifiers in logs. The second one is more general and can be used on more systems, but can lack precision. It is based on the frequencies of the actions and requires that every action appears frequently enough in traces. Other factors can be defined, for example, by computing the similarity of the actions that compare their common characters. Several string similarities could be used; we refer to [13] for the presentation and definition of some of them.

Two relations are defined from this coefficient, representing the notions of *strong* and *weak correlation* of actions. We say that two actions $a_1(\alpha_1), a_2(\alpha_2)$ have a *strong correlation*, denoted $a_1(\alpha_1)$ strong-corr $a_2(\alpha_2)$, if their *Correlation coefficient* is higher than a threshold X, defined by the user. We use a threshold because, for data and text mining, this notion often depends on the considered context. Similarly, we say that two actions $a_1(\alpha_1), a_2(\alpha_2)$ have a *weak correlation*, denoted $a_1(\alpha_1)$ weak-corr $a_2(\alpha_2)$, if their *Correlation coefficient* is lower than X. We denote *strong-corr(σ_i)*, a sequence of actions σ_i such that for all action $a_i(\alpha_i)$

of the sequence, $a_i(\alpha_i)$ strong-corr $a_{i+1}(\alpha_{i+1})$ holds. We also define the correlation between sequences of actions in order to determine if two sequences σ_1 and σ_2, such that *strong-corr(σ_1)* and *strong-corr(σ_2)*, are produced by the same component. Two sequences σ_1, σ_2 have a *weak correlation*, denoted σ_1 *weak-corr* σ_2, if the last action of σ_1 has a *weak correlation* with the first action of σ_2, meaning that these two sequences express the behavior of different components.

Definition 6 (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* $a_2(\alpha_2) \Leftrightarrow_{def} Corr(a_1(\alpha_1), a_2(\alpha_2)) \geq X$.
   $a_1(\alpha_1)$ *weak-corr* $a_2(\alpha_2) \Leftrightarrow_{def} Corr(a_1(\alpha_1), a_2(\alpha_2)) < X$.

2. *strong-corr*(σ_1) *iff* $\begin{cases} \sigma_1 = a(\alpha) \in \mathcal{L}, \\ \sigma_1 = a_1(\alpha_1) \dots a_k(\alpha_k)(k > 1) \in \mathcal{L}^*, \\ \forall (1 \leq i < k) : a_i(\alpha_i) \textit{ strong-corr } a_{i+1}(\alpha_{i+1}) \end{cases}$

3. σ_1 *weak-corr* σ_2 *iff* $\begin{cases} \sigma_2 = \epsilon, \\ \sigma_2 = a'_1 \dots a'_l \in \mathcal{L}^* \wedge (a_k \textit{ weak-corr } a'_1) \end{cases}$

With the help of these notions of *weak* and *strong correlation*, the procedure $Inspect$ given in Algorithm 2 splits a trace given as input into sub-sequences of actions, where each action of a sub-sequence has a *strong correlation* with the previous action of the sub-sequence, and each sub-sequence has a *weak correlation* with the previous sub-sequence.

Algorithm 2: Procedure Inspect

```
1 Procedure Inspect(t) : σ'_1 σ'_2 ... σ'_k is
2     Find the non-empty sequences σ'_1 σ'_2 ... σ'_k such that: t = σ'_1 σ'_2 ... σ'_k,
      strong-corr(σ'_i)_(1≤i≤k), (σ'_i weak-corr σ'_(i+1))_(1≤i≤k−1);
```

Let us illustrate the $Inspect$ procedure on a trace set $Traces(SUL)$ including the trace of Figure 3.3, and with a *Correlation coefficient* defined by the factor f_2, which computes the frequency of having successive actions in traces. The use of this Correlation coefficient on a trace set containing this trace reveals that the trace of Figure 3.3 can be separated into 7 sub-sequences, denoted σ_1 to σ_7, as the actions in the same sub-sequence are frequently consecutive.

The traces, as sequences of sub-sequences, are then taken one by one as input by the procedure $ExtractC$ that separates the sub-sequences that come from different components. The procedure $ExtractC$ is shown in Algorithm 3. Intuitively, the procedure extracts the sub-sequences that have a weak correlation with the first sub-sequence of the trace and replaces them by synchronization actions of the form $call\_C_i$ and $return\_C_i$ to model component calls, with C_i referring to a future LTS representing another component. The procedure takes a trace t,

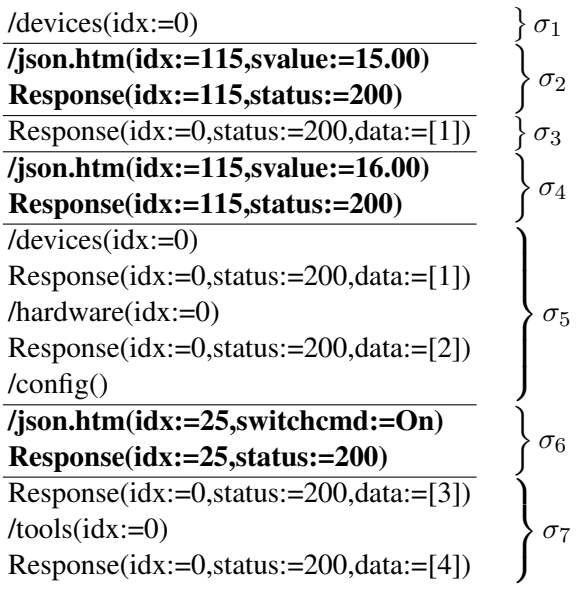


Figure 3.3: Example of a trace separated into sub-sequences.

splits it, and stores the resulting trace into a set T_c. The procedure $ExtractC$ tries to find the next sequence σ_j such that *strong-corr(σ_i.σ_j)* holds. The trace $t' = \sigma_{i+1}...\sigma_{j-1}$ (or $t' = \sigma_{i+1}...\sigma_k$ when there is no σ_j that has a strong correlation with σ_i), contains behaviors of other components as these sequences have a weak correlation with σ_i, so it is extracted. If this trace t' is composed of only one sub-sequence, then it is added to a new trace set T_n of $STraces$, surrounded by synchronization actions denoting that the component is called by another one. If it contains two or more sub-sequences, the procedure $ExtractC$ is recursively called with $ExtractC(t', T_n, STraces)$. The sequence t' is then removed from t and replaced by the actions $call\_C_n.return\_C_n$. The procedure continues to cover t by trying to find the first sub-sequence that has a strong correlation with σ_j, until all the sub-sequences of the trace are covered. Once all the sub-sequences of t are covered by the procedure, it checks if t represents the behavior of the root component and surrounds the trace with $call\_C_c$ and $return\_C_c$ if it is not the case, to express that the component is called by another one.

Figure 3.4 illustrates the functioning of the procedure $ExtractC$, with the trace of Figure 3.2. This trace was segmented into seven sub-sequences σ_1 to σ_7, such that *strong-corr(σ_i)*. We start at σ_1 ($i := 1$). The first sequence that is strongly 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 surrounded with the actions $call\_C2$ and $return\_C2$ to prepare the LTSs synchronization and is added to the set T_2. We go back to the trace t, at the sub-sequence σ_3 ($i := 3$). The same process is applied until the trace is covered. The algorithm

Algorithm 3: Procedure $ExtractC$

```
1  Procedure ExtractC(t = σ1σ2...σk, Tc, STraces): STraces is
2      i := 1;
3      while i < k do
4          n := |STraces| + 1;
5          Tn := {};
6          STraces := STraces ∪ {Tn};
7          σp is the prefix of t up to σi;
8          if ∃j > i: strong-corr(σi.σj) then
9              σj is the first sequence in σi ... σk such that strong-corr(σiσj);
10             t := σpσi.call_Cn return_Cn.σj ... σk;
11             if (j − i) > 2 then
12                 Extract(σi+1 ... σj−1, Tn);
13             else
14                 Tn := Tn ∪ {call_Cn.σi+1.return_Cn};
15             i := j;
16         else
17             t := σpσi.call_Cn return_Cn;
18             if (k − i) > 1 then
19                 Extract(σi+1 ... σk, Tn);
20             else
21                 Tn := Tn ∪ {call_Cn.σk.return_Cn};
22             i := k;
23     if c ≠ 1 then
24         t = call_Cc.t.return_Cc;
25     Tc := Tc ∪ {t};
26     return STraces;
```

found that σ_5 is the first sequence that is strongly correlated with σ_3. As previously, the sequence σ_4 is then extracted and replaced by the actions $call\_C3$ $return\_C3$, and the sequence σ_4 is surrounded with the actions $call\_C3$ and $return\_C3$ before being added to the set T_3. We go back to the trace t at the sub-sequence σ_5 ($i := 5$). Finally, the first sub-sequence that is strongly correlated with σ_5 is σ_7. The sequence σ_6 is extracted, replaced by the actions $call\_C4$ $return\_C4$, and surrounded with the actions $call\_C4$ and $return\_C4$ before being added to the set T_4. As σ_7 ($i := 7$) is the last sub-sequence of t, the trace was entirely covered and t becomes $\sigma_1.call\_C2$ $return\_C2.\sigma_3.call\_C3$ $return\_C3.\sigma_5.call\_C4$ $return\_C4.\sigma_7$. The trace t comes from $Traces(SUL)$, which means that t captures the behavior of a component that has not been called by another component. Hence, t is not surrounded by synchronization actions and is placed into the trace set T_1. At the end of this process, we get four trace sets, each composed of one trace, given in Figure 3.5.

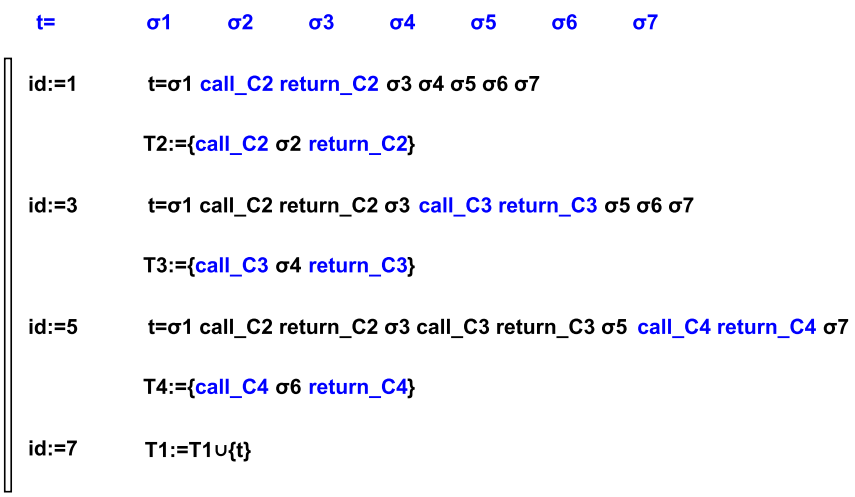


Figure 3.4: Example of procedure $ExtractC$ execution for COnfECt.

T_1 :	T_2 :
/devices(idx:=0)	call_C_2
call_C_2 return_C_2	/json.htm(idx:=115,svalue:=15.00)
Response(idx:=0,status:=200,data:=[1])	Response(idx:=115,status:=200)
call_C_3 return_C_3	return_C_2
/devices(idx:=0)	T_3 :
Response(idx:=0,status:=200,data:=[1])	call_C_3
/hardware(idx:=0)	/json.htm(idx:=115,svalue:=16.00)
Response(idx:=0,status:=200,data:=[2])	Response(idx:=115,status:=200)
/config()	return_C_3
call_C_4 return_C_4	T_4 :
Response(idx:=0,status:=200,data:=[3])	call_C_4
/tools(idx:=0)	/json.htm(idx:=25,switchcmd:=On)
Response(idx:=0,status:=200,data:=[4])	Response(idx:=25,status:=200)
	return_C_4

Figure 3.5: Example of results obtained by the *Trace Analysis & Extraction* step of COnfECt.

3.4.3 LTS Generation

The *LTS generation* step aims to produce the first models from the traces produced in $STraces$, during the previous step. Each subset of $STraces$ will form an LTS representing the behavior of a component. Given a subset T_1 in $STraces$, a trace of T_1 is lifted to the level of a LTS path. The LTS is obtained by joining the paths with a disjoint union on the state $q0$. COnfECt produces paths starting from the

initial state $q0$ and ending to a final state $q_k \neq q0$.

Definition 7 (LTS inference) *Let $T_1 \in STraces$ be a trace set. The LTS C_1 expressing the behaviors found in T_1 is the tuple $\langle Q, q0, \Sigma, \rightarrow \rangle$ where $q0$ is the initial state, q_k is a final state, and $Q, \Sigma, \rightarrow$ are defined by the following rule:*

$$\frac{t{=}a_1(\alpha_1)...a_k(\alpha_k)}{q0 \xrightarrow{a_1(\alpha_1)} q_1 ... q_{k-1} \xrightarrow{a_k(\alpha_k)} q_k}$$

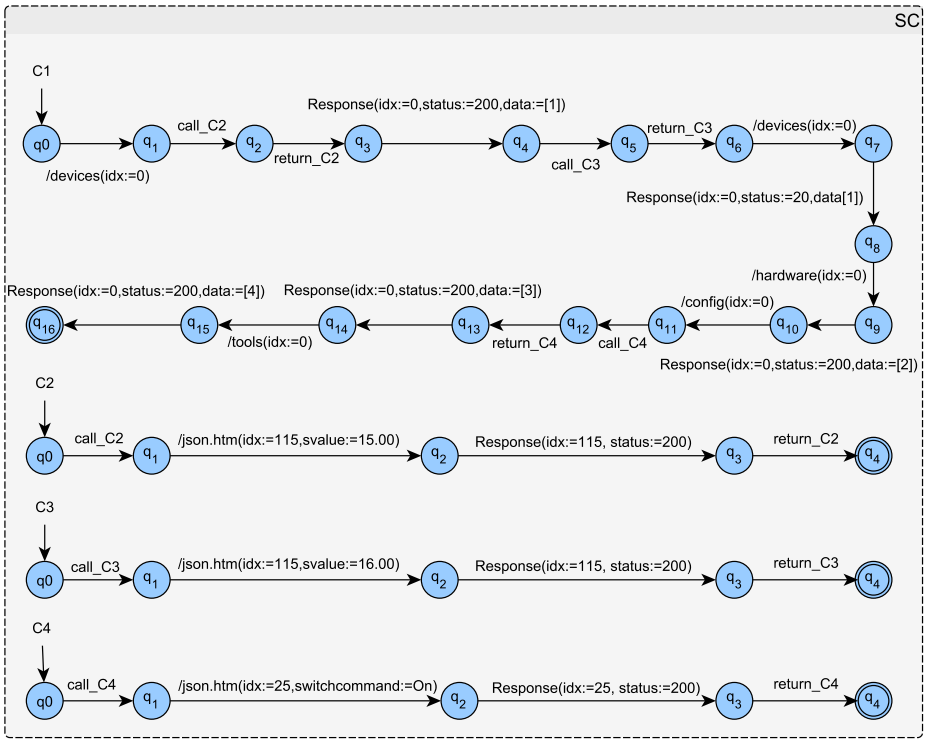


Figure 3.6: Example of models generated by the *LTS Generation* step of COnfECt.

Figure 3.6 shows the LTSs generated by this step. Each one of the four traces of Figure 3.5 leads to a model of the form of a chain, where the transitions represent the actions in the trace. For example, the trace T_2 leads to the model C_2 following the same action sequence: $call\_C_2$ $/json.htm(idx := 115, svalue := 15.00)$ $Response(idx := 115, status := 200)$ $return\_C_2$.

3.4.4 LTS synchronization

This step aims to modify the system of LTSs produced in the previous step in order to modify the synchronization of the components. 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 of the form $call\_C_i$ and $return\_C_i$. When the transition $q \xrightarrow{call\_C_2} q'$ is fired in the LTS C_1, we say that C_1 calls the component C_2. The component C_1 pauses its execution, while C_2 starts its execution from its initial state. When the transition $q \xrightarrow{call\_C_2} q'$ is fired in the LTS C_2, we say that C_2 is called. C_2 ends its execution when the transition $q \xrightarrow{return\_C_2} q'$ is fired in C_2, and C_1 can then continue its execution where it stopped it by firing the transition $q \xrightarrow{return\_C_2} q'$.

Algorithm 4: LTS synchronization strategies of COnfECt

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 synchronization* **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 synchronization* **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;
  return $kTail(k = 2, SC_f)$

Algorithm 4 summarizes the step *LTS synchronization* of COnfECt. COnfECt proposes three synchronization strategies representing three levels of generalization, presnted next.

Strict Strategy

This strategy aims to limit the over-generalisation of the models, i.e., producing models that express more behaviors than the system. This strategy is implemented

by lines 1 and 2 of Algorithm 4. In this strategy, the synchronization actions are not modified and strictly follow the behaviors in the traces, and a component calls another component only once. Finally, kTail [11] is applied to every model of the system in order to merge the states that have the same $kfuture$. Figure 3.7 shows the system of LTSs obtained after application of this strategy on the system of Figure 3.6. In this example, no states are merged with kTail because there are no states with the same $2future$.

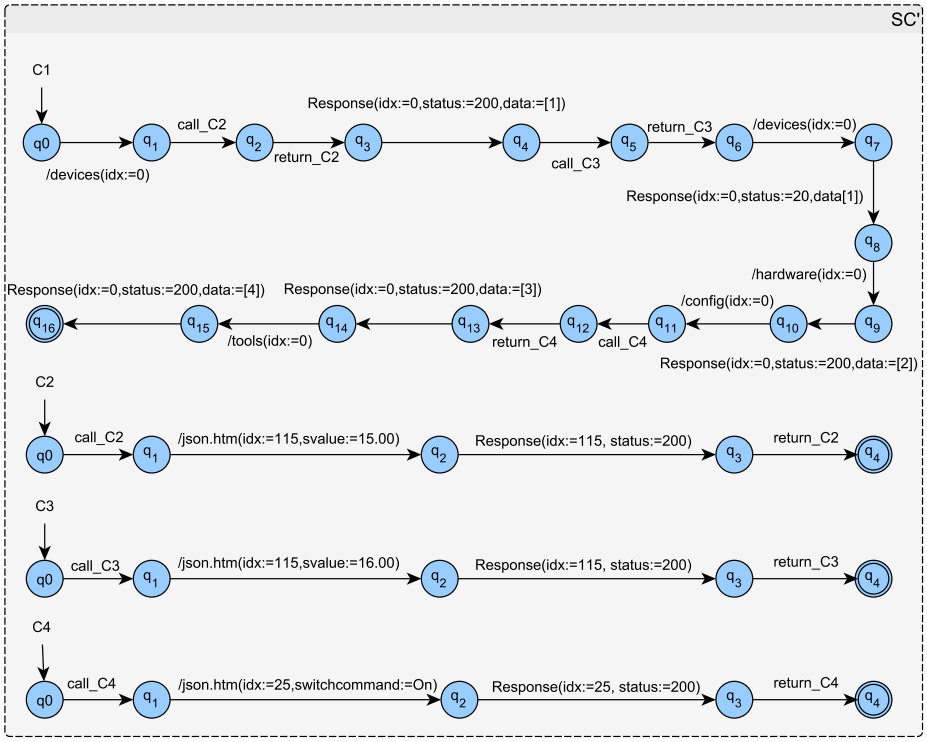


Figure 3.7: Example of result obtained with the Strict strategy.

Weak Strategy

This strategy aims to reduce the number of models produced and to allow repetitive component calls. This strategy is implemented in lines 3 to 16 of Algorithm 4. The step *LTS Generation* of COnfECt may have produced too many LTSs; this strategy tries to reduce their number by merging the similar LTSs that express behaviors of the same component. To do so, we use a *Similarity coefficient*, expressing the similarity between two LTSs. This *Similarity coefficient* is defined as a utility function, with factors defined by the user. Two LTSs are similar if their *Similarity coefficient* is higher than a threshold Y.

Definition 8 (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* $Y \in [0, 1]$.

Some examples of *Similarity coefficients* are given below. The first one expresses that two models are similar only if they contain actions that share the same component identifier. The second one computes the similarity between the label and parameters of the actions.

- $f'_1(C_1, C_2)$= 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 to the synchronized 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 [52]. We have chosen the Overlap coefficient because the action sets of two LTSs may have different sizes.

This *Similarity coefficient* is then used to classify the LTSs with the help of a hierarchical clustering technique. The LTSs in the same cluster are then merged with a disjoint union. As new LTSs are built, the synchronization actions are adapted according to the new LTSs (line 2); Moreover, with this strategy, the components are allowed to call several times in a row a component or not at all. This is represented in the model by replacing all the paths of the form $q_1 \xrightarrow{call\_C\ return\_C} q_3$, 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. Finally, kTail is called on every model of component to reduce their sizes. Figure 3.9 shows the system of LTSs we obtained by applying the Weak strategy to the system of Figure 3.6. By using the factor f'_2, the similarities found between the LTSs are shown in Figure 3.8. With a threshold of 0.5, the clustering method found that the LTSs $C2$ and $C3$ are similar. They are then merged in a new LTS called $C2C3$ in the final system of LTS SC'. In this example, kTail merges states with the same $2future$ in the model $C2C3$.

Strong Strategy

The last strategy of COnfECt also groups the similar LTSs and allows any component to call any other one at any moment. This strategy is implemented in lines 3 to 20 of Algorithm 4. When an LTS C_1 of a system SC can call any other LTS C_2 of SC at any of its states, we say that C_1 is *callable-complete*.

	$C1$	$C2$	$C3$	$C4$
$C1$	1	0.17	0.17	0.17
$C2$	0.17	1	0.67	0.33
$C3$	0.17	0.67	1	0.33
$C4$	0.17	033	0.33	1

Figure 3.8: Example of similarity matrix of LTSs of Figure 3.6.

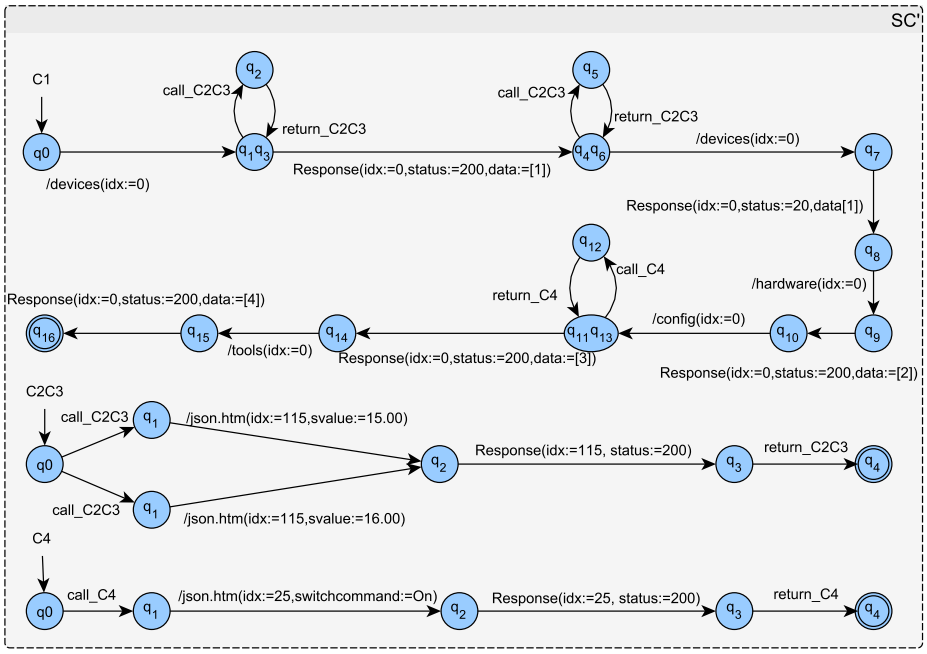


Figure 3.9: Example of result obtained with the Weak strategy.

Definition 9 (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 $\forall q \in Q_1, \forall C_2 \in C \setminus \{C_1\}, \exists q' \in Q_1 : q \xrightarrow{call\_C_2 return\_C_2} q'$.*

In this strategy, the models are hence grouped with the help of the *Similarity coefficient* and the clustering technique as in the Weak strategy. The main difference with the Weak strategy concerns the synchronizations actions. After the transformation of the paths of the form $q_1 \xrightarrow{call\_C\ return\_C} q_3$ into loops, the models are then completed by adding in every state q_i a loop $q_i \xrightarrow{call\_C_n\ return\_C_n} q_i$ for every component C_n, making them callable-complete. Finally, kTail is called again. Figure 3.10 shows the models obtained by applying the Strong strategy to the system of LTSs in Figure 3.6. To make it more readable, in this figure all the loops $q_i \xrightarrow{call\_C_n\ return\_C_n} q_i$ are abstracted into a single loop $q_i \xrightarrow{call\_C\ return\_C} q_i$.

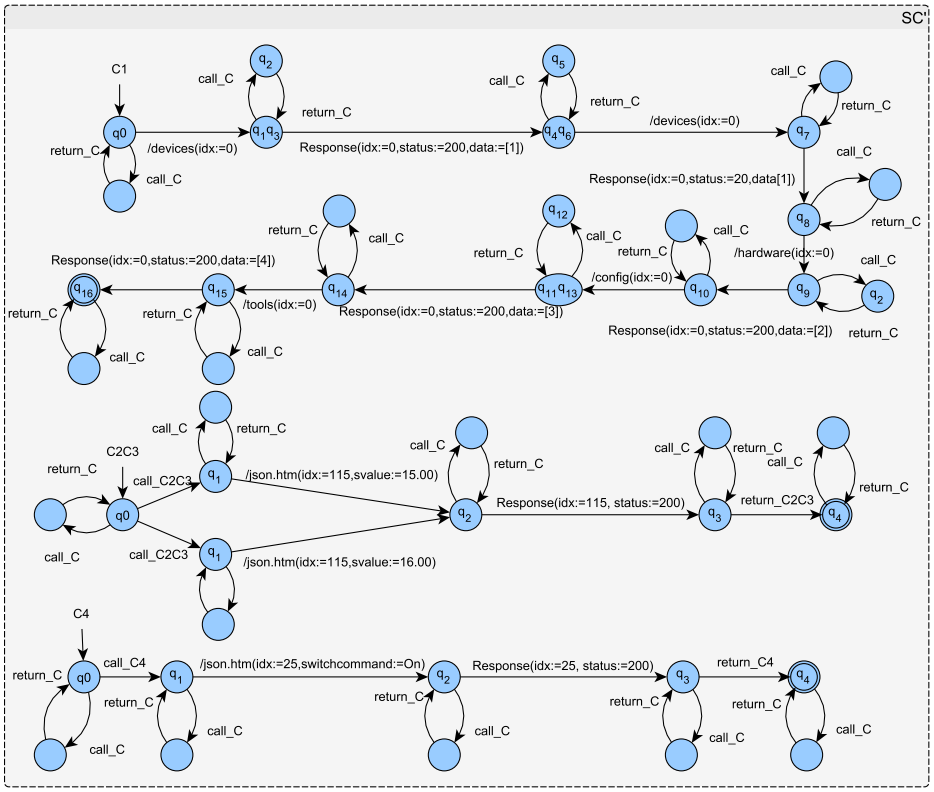


Figure 3.10: Example of result obtained with the Strong strategy.

3.5 Description of ASSESS

3.5.1 Trace Formatting

This step is identical to the one described in Section 3.4.1. As with COnfECt, at the end of this step, we assume having a trace set denoted $Traces(SUL)$, which gathers traces of the form $a_1(\alpha_1)...a_k(\alpha_k)$.

3.5.2 Trace Analysis and Extraction

As with COnfECt, this step aims to analyze the traces of $Traces(SUL)$ and produce a new set $STraces$ that contains trace sets representing only one component. With ASSESS, we assume that components are identified in the action with a component identifier. ASSESS covers the traces of $Traces(\mathrm{SUL})$ and relies on these identifiers to detect component calls and gather the traces related to each component in separate trace sets.

Definition 10 (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 $ID : \mathcal{L} \rightarrow P$, 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 $a_1(\alpha_1)a_2(\alpha_2)\ldots a_k(\alpha_k)$ is given by the mapping $ID_s : \mathcal{L}^* \to P$. $ID_s(a_1(\alpha_1)a_2(\alpha_2)\ldots a_k(\alpha_k)) =_{def}$
$\begin{cases} \alpha' \textit{ iff } \forall a_i \notin \{call, return\} : ID(a_i(\alpha_i)) = \alpha'(1 \leq i < k) \\ \{\} \textit{ otherwise}. \end{cases}$

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

Algorithm 5: Component Trace Detection

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

ASSESS proceeds as shown in Algorithm 5. For each trace of $Traces(SUL)$, it first detects $ID(a_1(\alpha_1))$, the identifier of the first action of the trace. Then, it separates the actions that come from different components, with the help of the procedure $ExtractA$, that produces a set of traces called $Traces$, where each trace contains the behavior of a single component. Once all the traces are covered, all the traces of $Traces$ are grouped with the help of the procedure $GroupById$, which produces a new set $STraces = \{T_1, T_2, \ldots, T_n\}$, which contains a trace set T_i for each component C_i.

The procedure $ExtractA(t = a_1(\alpha_1)a_2(\alpha_2)\ldots a_k(\alpha_k), T, id)$ is given in Algorithm 6. It takes as input the trace t, the set T that will contain the traces of components, and id the identifier of the first action of the trace $ID(a_1(\alpha_1))$. The procedure covers the actions of t and checks the identifiers. If it detects an action with an identifier n that is different from $newid$, we assume that a new component is called, and the action has to be extracted. In that case, the procedure searches for the sub-sequence $t_n = a_{i+1}(\alpha_{i+1})...a_{j-1}(\alpha_{j-1})$ composed of actions with an identifier different from $newid$ and replaces it by the synchronization actions $call\_C_n.return\_C_n$, representing that a new component n is called. If t_n contains more than one action, the procedure $ExtractA$ is recursively called to cover the actions and build new traces. Otherwise, t_n is surrounded by the synchronization actions $call\_C_n$ and $return\_C_n$. If there already exists a trace t_2 in T with $ID(t_2) = ID(t_n)$, t_n is concatenated to t_2. If it is not the case, t_n is added to the trace set T. Once the procedure finishes to cover the trace t, we obtain a new trace t' containing only actions of one component and synchronization actions. It then checks if the component is called by comparing the identifier of t with id given

Algorithm 6: Procedure $ExtractA$

```
1  Procedure ExtractA(t = a_1(α_1)a_2(α_2)...a_k(α_k), T, id): T is
2      newid := Identifier(a_1(α_1));
3      t' := a_1(α_1); a_{k+1}(α_{k+1}) = ε; i := 1;
4      while i < k do
5          n := ID(a_{i+1}(α_{i+1}));
6          if n == newid then
7              t' := t'.a_{i+1}(α_{i+1});
8              j := i + 1;
9          else
10             find smallest j > i such that ID(a_j(α_j)) == newid or
                j := k + 1;
11             t' := t'.call(n)return(n).a_j(α_j);
12             if (j − i) > 2 then
13                 Extract(a_{i+1}(α_{i+1})...a_{j−1}(α_{j−1}), T, id);
14             else
15                 t_n := call(n).a_{i+1}(α_{i+1}).return(n);
16                 if ∃t_2 ∈ T : ID(t_2) == n then
17                     t_n := t_2.t_n; T := T \ {t_2};
18                 T := T ∪ {t_n};
19         i := j;
20     if newid ≠ id then
21         t' := call(newid).t'.return(newid);
22     if ∃t_2 ∈ T : ID(t_2) == newid then
23         t' := t_2.t'; T := T \ {t_2};
24     T := T ∪ {t'};
25     return T;
```

as input, and if it is different, it surrounds it with synchronization actions denoting that the component is called. Finally, if there already exists in T a trace t_2 with $ID(t_2) = ID(t')$, t' is concatenated to t_2. If it is not the case, t' is added to the trace set T.

Figure 3.12 shows an example of traces obtained after the execution of the procedure $ExtractA$ on the trace given in Figure 3.11. First, the procedure detects that the actions $a_2(\alpha_2)$ and $a_3(\alpha_3)$ do not have the same identifier as the first one $a_1(\alpha_1)$, $idx := 0$. They are extracted and replaced by the synchronization actions $call\_C_2$ and $return\_C_2$. The sub-sequence $a_2(\alpha_2)a_3(\alpha_3)$ is put in a new trace T_2 surrounded by $call\_C_2$ and $return\_C_2$. Next, the procedure finds that the actions $a_5(\alpha_5)a_6(\alpha_6)$ come from another component and are extracted from the trace. This time, as there already exists a trace T_2 with the same identifier as these actions, the sub-sequence $a_5(\alpha_5)a_6(\alpha_6)$ surrounded by $call\_C_2$ and $return\_C_2$ is concatenated to T_2. Finally, the procedure detects that the sub-

$a_1(\alpha_1) =$ /devices(idx:=0)
$a_2(\alpha_2) =$ /json.htm(idx:=115,svalue:=15.00)
$a_3(\alpha_3) =$ Response(idx:=115,status:=200)
$a_4(\alpha_4) =$ Response(idx:=0,status:=200,data:=[1])
$a_5(\alpha_5) =$ /json.htm(idx:=115,svalue:=16.00)
$a_6(\alpha_6) =$ Response(idx:=115,status:=200)
$a_7(\alpha_7) =$ /devices(idx:=0)
$a_8(\alpha_8) =$ Response(idx:=0,status:=200,data:=[1])
$a_9(\alpha_9) =$ /hardware(idx:=0)
$a_{10}(\alpha_{10}) =$ Response(idx:=0,status:=200,data:=[2])
$a_{11}(\alpha_{11}) =$ /config(idx:=0)
$a_{12}(\alpha_{12}) =$ /json.htm(idx:=25,switchcmd:=On)
$a_{13}(\alpha_{13}) =$ Response(idx:=25,status:=200)
$a_{14}(\alpha_{14}) =$ Response(idx:=0,status:=200,data:=[3])
$a_{15}(\alpha_{15}) =$ /tools(idx:=0)
$a_{16}(\alpha_{16}) =$ Response(idx:=0,status:=200,data:=[4])

Figure 3.11: Example of a trace that can be taken as input by the procedure $ExtractA$ of ASSESS. The identifier idx expresses which component produced the action.

sequence, $a_12(\alpha_12)a_13(\alpha_13)$ comes from another component and is extracted in a new trace in the set T_3. In comparison to the result obtained with the *Trace Analysis & Extraction* step of COnfECt in Figure 3.5, ASSESS produces only 3 traces, one for each component, whereas COnfECt produces 4 traces. ASSESS has already grouped all the actions that come from the same component, whereas COnfECt only separated those that come from different components and later need to group them.

Finally, the traces are partitioned by the procedure $GroupById(Traces)$, which returns the set $STraces$, such that every subset holds traces having the same component identifier. We partition $Traces$ by defining the trace equivalence relation $\sim_{id}$ and by extracting the equivalence classes of $Traces$ for $\sim_{id}$. Let $\sim_{id}$ 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 $STraces = Traces/\sim_{id}$. At the end of this step, we obtain the set $Straces = \{T_1, T_2, ..., T_n\}$, where T_i represents the behavior of only one component.

3.5.3 LTS Generation

The *LTS generation* step aims to produce the first models of components from the traces produced in $STraces$. Each subset of $STraces$ will form an LTS representing the behavior of a component. Given a subset T_1 in $STraces$, a trace of T_1 is transformed into a LTS cycle. The cycles are then joined with a disjoint union. by the state $q0$ to form the first LTSs. This step is very similar to the one of COnfECt

T_1	T_2
/devices(idx:=0)	call_C_2
call_C_2 return_C_2	/json.htm(idx:=115,svalue:=15.00)
Response(idx:=0,status:=200,data:=[1])	Response(idx:=115,status:=200)
call_C_2 return_C_2	return_C_2
/devices(idx:=0)	call_C_2
Response(idx:=0,status:=200,data:=[1])	/json.htm(idx:=115,svalue:=16.00)
/hardware(idx:=0)	Response(idx:=115,status:=200)
Response(idx:=0,status:=200,data:=[2])	return_C_2
/config(idx:=0)	T_3
call_C_3 return_C_3	call_C_3
Response(idx:=0,status:=200,data:=[3])	/json.htm(idx:=25,switchcmd:=On)
/tools(idx:=0)	Response(idx:=25,status:=200)
Response(idx:=0,status:=200,data:=[4])	return_C_3

Figure 3.12: Example of result obtained by the procedure $ExtractA$ of ASSESS.

in 3.4.3. The only difference is that ASSESS produces cyclic paths, starting and ending at the initial state $q0$, whereas COnfECt produces paths starting from the initial state $q0$ and ending at a final state $q_k \neq q0$.

Definition 11 (LTS inference) *Let $T_1 \in STraces$ be a trace set. The LTS C_1 expressing the behaviors 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:*

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

Figure 3.13 shows an example of models generated by the *LTS Generation* step of ASSESS from the set $STraces$ of Figure 3.12. For each subset of Figure 3.12, a model representing with the form of a cycle is produced following the same behavior. For example, the set T_3 is composed of one trace, and leads to the LTS C_3 following the same behavior $call\_C_2\ /json.htm(idx := 115, svalue := 15.00)\ Response(idx := 115, status := 200)\ return\_C_2$. In comparison to the LTSs generated by COnfECt in Figure 3.6, the system of LTSs given by COnfECt contains more LTSs than the one given by ASSESS. COnfECt has not yet grouped the LTSs that come from the same component (here C_2 and C_3), and will do it in the step *LTS synchronization* next, whereas ASSESS has already grouped them during the *Trace Analysis and Extraction* step. Moreover, we can see that the LTSs produced by the *LTSs Generation* step of ASSESS are cyclic. We believe that a cyclic model is a better representation for IoT components, that generally have short and repeatable behaviors.

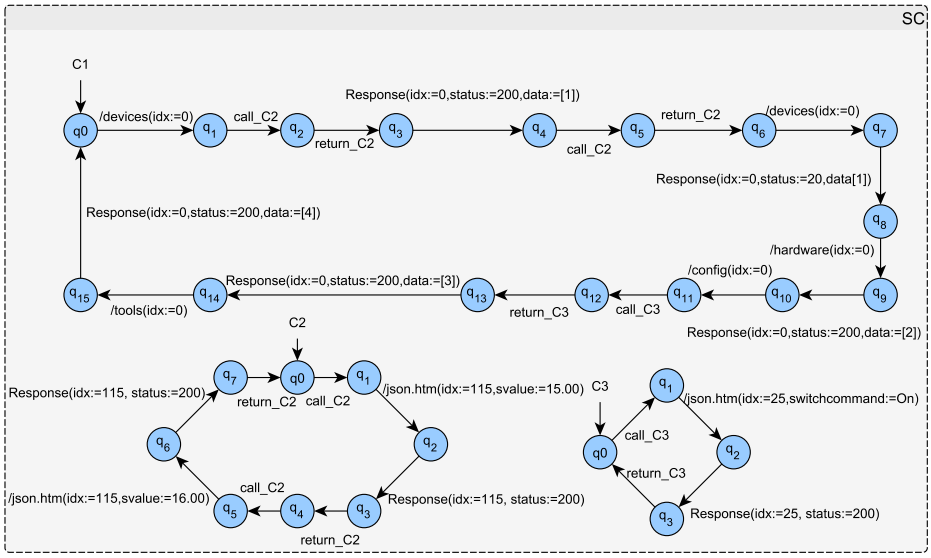


Figure 3.13: Example of models generated by the *LTS Generation* step.

3.5.4 LTS synchronization

This step generalizes the system of LTSs according to the strategy chosen by the user. That represents the architecture of the system. Algorithm 7 summarizes the step *LTS synchronization* of ASSESS. ASSESS proposes two strategies. The *Loose-coupling* strategy represents a system where the components interact with each other, and the *Decoupling* strategy is where the components run in parallel.

Algorithm 7: LTS synchronization Strategies of ASSESS

Procedure *Loose-coupling*$(SC = \langle S, \{C_1, C_2, \ldots, C_n\}\rangle) : SC_1$ **is**
  foreach $C_i = \langle Q, q0, \Sigma, \rightarrow\rangle \in C$ **do**
    foreach $q_1 \xrightarrow{call(\sigma)return(\sigma)} q_2$ **do**
      merge q_1 and q_2;
    $C'_i := kTail(k = 2, C_i)$;
  return $\langle S, \{C'_1, C'_2, \ldots, C'_n\}\rangle$
Procedure *Decoupling*$(SC = \langle S, \{C_1, C_2, \ldots, C_n\}\rangle) : SC_2$ **is**
  foreach $C_i = \langle Q, q0, \Sigma, \rightarrow\rangle \in C$ **do**
    $C_i :=$ hide S in C_i;
    $C_i := \tau$-reduce C_i;
    $C'_i := kTail(k = 2, C_i)$;
  return $\langle S, \{C'_1, C'_2, \ldots, C'_n\}\rangle$

Loose-coupling Strategy

This strategy is implemented by the procedure *Loose-coupling* of Algorithm 7. The synchronization actions are kept in the model, but in order to generalize the model, the transitions of the form $q_1 \xrightarrow{call\_C} q_2\ q_2 \xrightarrow{return\_C} q_3$, are 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. Finally, kTail is applied to every model in order to reduce their size.

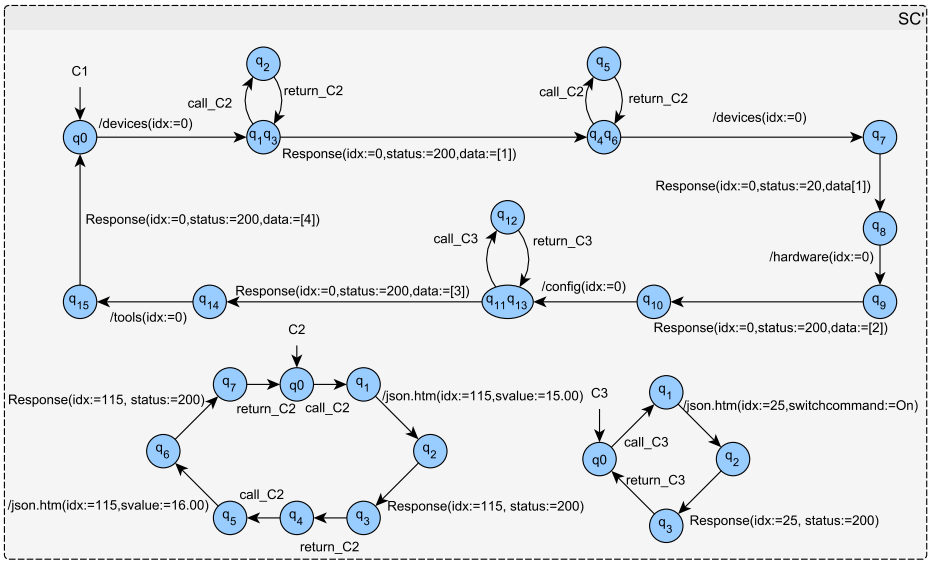


Figure 3.14: Example of result obtained with the *Loose-coupling* strategy.

Figure 3.14 shows the models obtained by applying the *Loose-coupling* strategy on the system of LTSs of Figure 3.13. This strategy is similar to the *Weak* strategy of COnfECt. Both strategies replace the synchronization actions by loops, to accept consecutive calls. The main difference lies in the fact that ASSESS builds cyclic LTSs.

Decoupling Strategy

This strategy aims to produce a system of LTSs representing a system where the components do not depend on each other. It is implemented by the procedure *Decoupling* of Algorithm 7. The synchronization actions are hidden in the models, the LTS operator $hide$ that replaces all synchronization actions by a non-observable action τ. The models are then reduced with the procedure τ-$sreduce$ that removes the transitions labeled with τ.

Definition 12 (τ-reduction) *Let $C_1 = \langle Q_1, q0_1, \Sigma, \rightarrow_1$ be a LTS. τ-reduction $C_1 =_{def} \langle Q_2, q0_2, \Sigma, \rightarrow_2)$ where $Q_2, q0_2, \rightarrow_2$ are the minimal sets satisfying the following inference rules:*

$$\frac{q_1 \xrightarrow{a(\alpha)} q_2}{q_1 \xrightarrow{a(\alpha)} 2(q_2)} \quad \frac{q_1 \xrightarrow{a(\alpha)} q_2 \xrightarrow{\tau \ldots \tau} q_3}{q_1 \xrightarrow{a(\alpha)} 2(q_2 q_3)} \quad \frac{q_1 \xrightarrow{\tau \ldots \tau} q_2 \xrightarrow{a(\alpha)} q_3}{(q1q_2) \xrightarrow{a(\alpha)} 2(q_2)}$$

kTail is finally applied on LTSs in order to reduce their size.

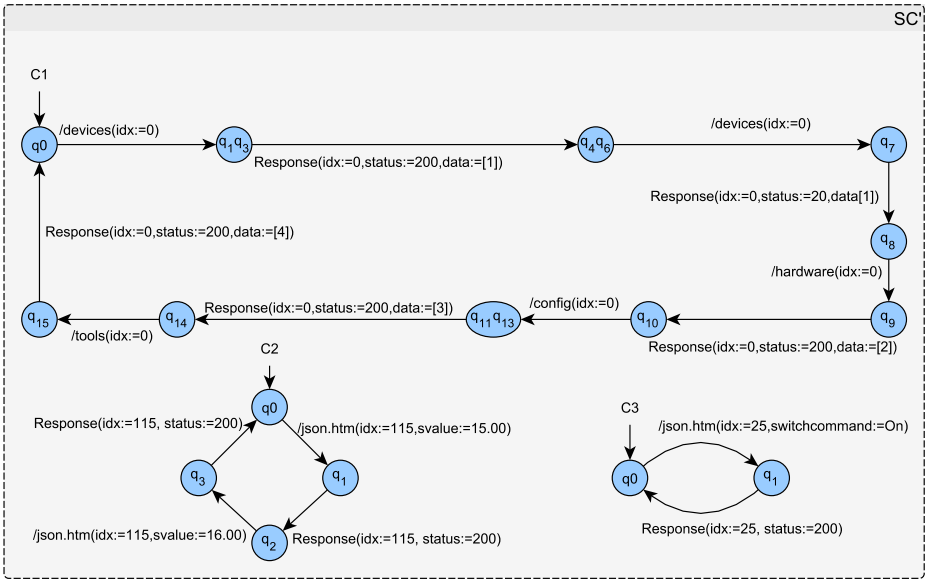


Figure 3.15: Example of result obtained with the *Decoupling* strategy.

Figure 3.15 shows the models we obtained by applying the *Decoupling* strategy on the system of LTSs in Figure 3.13.

3.6 Evaluation

Each method was implemented in order to evaluate them on real IoT systems. The first step of both methods is implemented in a separate tool called TFormat(https://github.com/Elblot/TFormat), which takes a log that can be collected on the components or with the help of a sniffer, and extracts formatted traces using regular expressions. The three other steps of ASSESS[2] and COnfECt[3] are implemented in separate tools. They take as inputs a set of formatted traces and a strategy, to produce a system of LTSs. These tools include an implementation of kTail, that will also be used in order to compare the results obtained with both methods and kTail. We used these tools to answer the following research questions:

- RQ1 (Component Detection) : Does COnfECt and ASSESS succeed to infer a model for each component of the system?

[2]https://github.com/Elblot/ASSESS
[3]https://github.com/Elblot/COnfECt2.0

- RQ2 (Model Readability) : Are the models generated by COnfECt and ASSESS more readable than the one generated by kTail?
- RQ3 (Rate of valid traces accepted): Do the models generated by COnfECt and ASSESS accept correct behaviors of the system?
- RQ4 (Rate of invalid traces accepted): Do the models generated by COnfECt and ASSESS reject incorrect behaviors of the system?
- RQ5 (Scalability) : how COnfECt and ASSESS scale with the size of the trace set given as input?

3.6.1 Setup

The following configurations are configurations designed to follow the hypothesis of both methods. These new experiments are exclusive to the thesis; further experiments are available in the papers presenting COnfECt [49] and ASSESS [50], where each method is evaluated independently.

First, 6 IoT systems were produced from different combinations based upon a gateway, 3 sensors, and 2 actuators, denoted $Conf1$ to $Conf6$. The sensors are temperature sensors and motion sensors. The gateway calls the sensors to obtain data, then calls the actuators (light bulbs) according to the data sent by the sensors. Another configuration is built from 8 sensors called by a gateway denoted $Conf7$. With every configuration, the gateway is the root component of the system. Three last configurations were constructed to compare the different correlation and similarity factors of COnfECt, denoted $Conf8$ to $Conf10$, composed of a gateway that calls a sensor in $Conf8$, $aNTPserverin$Conf9, and both in $Conf10$. The factors f_1, f_1' and the factors f_2, f_2' can all be applied for these three configurations.

A log file is extracted from the configuration $Conf1$ to $Conf10$, where identifiers are present in the events. These log files can then be taken as input by both methods as they meet the hypotheses of both methods. These logs are first given to the tool TFormat with a set of 12 regular expressions to extract formatted traces. The experimentations were performed on a desktop computer with 1 Intel(R) CPU i5-7500 @ 3.4GHz and 16GB RAM.

3.6.2 RQ1: Component Detection

Procedure: In order to answer this research question, we produced the models for every configuration using both methods with all strategies and kTail. We manually analyzed the systems and compared the number of real components with the number of LTSs generated by the methods. The factors used by COnfECt for $Conf1$ to $Conf7$ are the correlation factor f_1 and the similarity factors f_1', based on the identifier present in the events in the traces, we do not succeed in using the factors f_2, f_2', based on the frequency of the events in these configurations. However, as the factors f_1, f_1' and the factors f_2, f_2' can both be used for $Conf8$, the

second configuration of factors was also used in this configuration in order to illustrate the effects of different thresholds on the precision of the component detection.

Configuration	real number of components	COnfECt Strict	COnfECt Weak	COnfECt Strong	ASSESS Loose-coupling	ASSESS Decoupling	kTail
$Conf1$ $f_1 \geq 1$ $f_1' \geq 1$	6	847	6	6	6	6	1
$Conf2$ $f_1 \geq 1$ $f_1' \geq 1$	5	1108	5	5	5	5	1
$Conf3$ $f_1 \geq 1$ $f_1' \geq 1$	4	613	4	4	4	4	1
$Conf4$ $f_1 \geq 1$ $f_1' \geq 1$	5	1257	5	5	5	5	1
$Conf5$ $f_1 \geq 1$ $f_1' \geq 1$	6	1111	6	6	6	6	1
$Conf6$ $f_1 \geq 1$ $f_1' \geq 1$	4	1007	4	4	4	4	1
$Conf7$ $f_1 \geq 1$ $f_1' \geq 1$	9	100	9	9	9	9	1
$Conf8$ $f_1 \geq 1$ $f_1' \geq 1$	2	42	2	2	2	2	1
$Conf8$ $f_2 \geq 0.5$ $f_2' \geq 0.5$	2	54	2	2	2	2	1
$Conf8$ $f_1 \geq 1$ $f_1' \geq 1$	2	44	2	2	2	2	1
$Conf9$ $f_2 \geq 0.5$ $f_2' \geq 0.5$	2	88	2	2	2	2	1
$Conf10$ $f_1 \geq 1$ $f_1' \geq 1$	3	164	3	3	3	3	1
$Conf10$ $f_2 \geq 0.25$ $f_2' \geq 0.5$	3	184	4	4	3	3	1
$Conf10$ $f_2 \geq 0.5$ $f_2' \geq 0.5$	3	249	5	5	3	3	1
$Conf10$ $f_2 \geq 0.5$ $f_2' \geq 0.4$	3	249	3	3	3	3	1

Table 3.1: Number of LTSs generated for each configuration.

Results: Table 3.1 shows the number of generated LTSs. As expected, only one model is produced by kTail for each configuration, as kTail does not separate the

behavior of the components. COnfECt using the Strict strategy infers too many components because the LTSs are not merged during the *LTSs synchronization* step, meaning that the behaviors of a component can be modeled with several LTSs. The Strict strategy cannot be used to detect the correct number of components. However, COnfECt with the Weak and Strong strategies always succeeded in finding the correct number of components when the factors f_1, f_1' are used. It is not surprising that using a component identifier available in the log leads to the correct detection of components. Similarly, we can see that ASSESS always finds the right number of components no matter the strategy used. It is once again not surprising, as ASSESS uses component identifiers to detect the different components in the system.

However, when the factors f_2, f_2' are used with COnfECt, we can see that different thresholds lead to different results. We first observed in the 14th line, using $f_2 \geq 0.25$ $f_2' \geq 0.5$, that some traces in $STraces$ produced by the step *Trace Analysis and Extraction* contain events that come from different components, meaning that the step does not separate enough the traces. The threshold of the correlation factor is then increased to $f_2 \geq 0.5$ in order to cut more the traces. We then observed that too many LTSs were produced using the Weak and Strong strategies, as some LTSs representing the same component were not merged during the *LTS synchronization* step. So in order to ease the merging of the LTSs, the similarity threshold is then reduced to $f_2' \geq 0.4$. Finally, the good number of components were detected using the thresholds $f_2 \geq 0.5$ $f_2' \geq 0.4$, and a manual analysis of the LTSs confirmed that each one contains only events of one component.

This experiment tends to show that the methods can detect the correct number of components of the system. In the case of COnfECt it requires that the correct factors and thresholds are chosen by the user.

3.6.3 RQ2: Model Readability

Procedure: In order to answer this research question, we compared the sizes of the models generated by each method using every strategy with kTail. The factors used by COnfECt are the factors f_1, f_1', based on the component identifiers present in the events in the traces.

Results: Table 3.2 gives the number of states and transitions of the generated LTSs. This table also shows the number of states and transitions we obtain if we hide the synchronization actions, denoted by $(hide)$ in the table. We can first see that COnfECt with the Strict strategy produces a huge system of LTSs, with an increase of 6337.38% of the states in comparison with the LTSs generated with kTail. This strategy produces a lot of small LTSs composed of generally 2 or 3 events, and some synchronization actions. As these LTSs are small, few states are merged during the synchronization step, leading to a less general system of LTSs and so, a bigger system of LTSs. The Weak and Strong strategies generate bigger models than kTail, with a number of states increased by respectively 96.02%

		$Conf1$	$Conf2$	$Conf3$	$Conf4$	$Conf5$	$Conf6$	$Conf7$	$Conf8$	$Conf9$	$Conf10$
COnfECt Strict strategy	states	7504	9698	5751	9874	8364	8421	881	366	400	1382
	transitions	7150	8927	4851	9290	7824	8045	891	390	430	1416
COnfECt Strict strategy ($hide$)	states	4107	5828	2905	5532	4538	5049	566	234	261	857
	transitions	3344	4499	2283	4266	3384	4021	495	226	258	766
COnfECt Weak strategy	states	265	139	195	151	231	121	169	63	84	158
	transitions	602	231	397	265	453	206	266	126	156	352
COnfECt Weak strategy ($hide$)	states	133	79	112	86	121	71	56	53	69	120
	transitions	294	137	223	165	281	125	145	96	114	238
COnfECt Strong strategy	states	123	70	112	89	92	79	118	52	60	78
	transitions	280	144	286	184	204	166	214	124	137	193
COnfECt Strong strategy ($hide$)	states	48	31	39	38	29	31	32	32	37	43
	transitions	83	55	84	63	54	58	50	61	59	82
ASSESS Loose-coupling strategy	states	229	142	213	157	239	128	170	62	81	162
	transitions	470	255	442	295	508	230	279	133	139	385
ASSESS Loose-coupling strategy ($hide$)	states	127	74	107	80	115	66	76	43	59	94
	transitions	292	137	237	163	277	122	137	93	112	231
ASSESS Decoupling strategy	states	64	32	53	42	39	39	55	33	35	44
	transitions	115	69	130	85	95	86	99	69	61	96
kTail	states	111	60	86	78	93	65	107	43	59	92
	transitions	231	128	181	154	204	125	164	92	116	229

Table 3.2: Number of states and transitions of the model generated by every method and strategy in normal condition and if the synchronization actions are hidden (call hidden).

and 12.60%. The main reason is the synchronization actions added in the LTSs during the execution of COnfECt. The observations are similar for the Loose-coupling strategy of ASSESS, that generates bigger LTSs where the states number is increased by 96.38%. However, the Decoupling strategy of ASSESS does not add synchronization actions in the LTSs, leading to the reduction of the number of states by 42.74% with this strategy.

The synchronization actions are useful to understand the combination of components and are important for the composition of LTSs, but they are not necessary and can be hidden if we want to concentrate on the behaviors of the components only. If these synchronization actions are hidden, the Weak strategy of COnfECt increases the states number by only 12.60%, and the Strong strategy reduces it by 52.82%. With ASSESS using the Loose-coupling strategy, the number of state is increased only by 4.44%. With the Loose-coupling strategy of ASSESS and the Weak strategy of COnfECt, we obtain models whose sizes are similar to the sizes of the LTS generated by kTail, but the models are separated into smaller LTSs. We believe that the models generated by ASSESS and COnfECt are still more readable as they are separated into smaller LTSs.

In summary, the models generated by the weak strategy of COnfECt and the Loose-coupling strategy of ASSESS give similar results. It is not surprising as the Loose-coupling strategy of ASSESS and the Weak strategy of COnfECt are built to produce a model with the same level of abstraction. Similarly, the Decoupled strategy of ASSESS and the Strong strategy of COnfECt when the synchronization actions are hidden also give similar results, as these two strategies are also designed to produce a model with the same level of abstraction.

3.6.4 RQ3: Rate of valid traces accepted

Procedure: In order to answer this research question, we separated the set of traces of each configuration with a ratio of approximately 70-30%. The first 70% were used to produce models with COnfECt, ASSESS, and kTail. Then, the second part was considered as valid traces of the system that will be replayed on the LTSs in order to measure the rate of valid traces not used for generating the models that are accepted by these models. The factors used by COnfECt to compare it with the other methods are the factors f_1, f_1'. Other experimentations were done on $Conf8$ to $Conf10$ with COnfECt, using the factors f_1, f_1' and f_2, f_2', in order to show the effect of the factors on the model precision.

Results: Figure 3.16 shows the rate of valid traces accepted by the models generated by every method and strategy for each configuration. The strategies of COnfECt and ASSESS lead to different results. The Strong, Weak and Strict strategies of COnfECt provide models that accept, respectively, an average of 94.72%, 88.40%, and 80.74%. These results were expected as these strategies were designed to express three levels of model generalization. The Weak strategy generalizes more the LTSs than the Strict strategy, as it allows components to call multiple

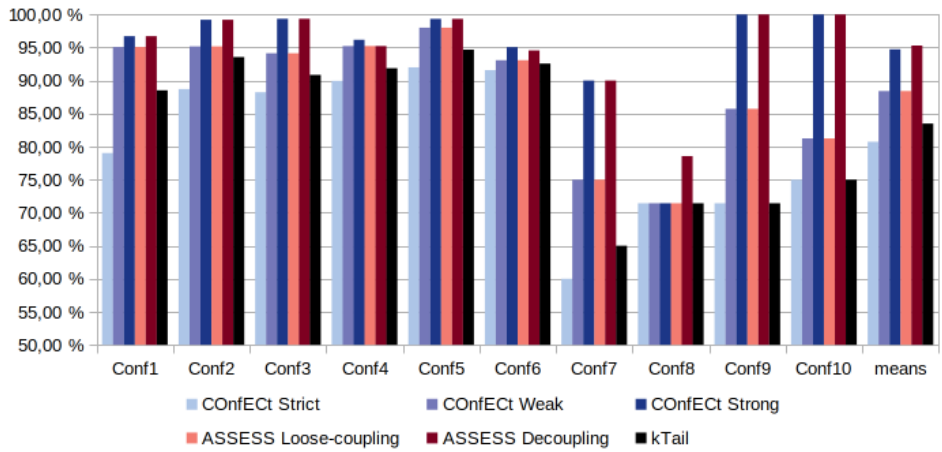


Figure 3.16: Rate of valid traces accepted by the models generated.

times other components, and the Strong strategy generalizes even more by allowing components to call the others at any time. All the strategies, except the Strict strategy, produce LTSs that accept more valid traces than the one generated from kTail, with an average of 83.49% of valid traces accepted. We observed that the Strict strategy of COnfECt splits the system of LTSs into many little LTSs, and consequently, fewer states are merged by the kTail algorithm, leading to a less generalized model. For ASSESS, models generated by the Decoupling strategy accept more valid traces, with 95.29% of accepted traces, against 88.40% for those generated by the Loose-coupling strategy. Once again, these results were expected, as the Decoupling strategy generalizes more the model than the Loose-coupling strategy, as the components can act in parallel and do not need to be called by the others.

We can see in Figure 3.16 that the Weak strategy of COnfECt and the Loose-coupling strategy of ASSESS obtained the same results. The two strategies are very similar, so this is not surprising. However, even if the Strong strategy of COnfECt and the Decoupling strategy of ASSESS are similar, we can see a little difference between their results. The models generated by ASSESS with the Decoupling strategy accept a little fewer traces than the ones generated by COnfECt with the Strong strategy. That is because during the Trace Extraction step, COnfECt produces new traces each time it finds a new behavior, where ASSESS concatenates the behaviors that come from the same components. That difference makes the COnfECt over-generalise a little more the model, as a split behavior of a component in the traces will also be split in two distinct behaviors in the models.

In summary, both ASSESS and COnfECt seem to have a similar rate of valid traces accepted, and the models generated by COnfECt and ASSESS accept more valid traces than the ones generated by kTail, except for the Strict strategy of COnfECt, which produces a model that accepts fewer valid traces than kTail.

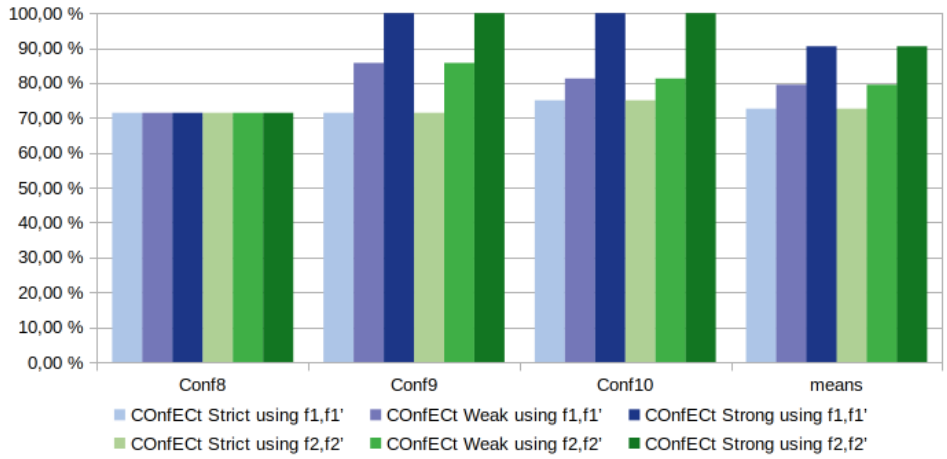


Figure 3.17: Rate of valid trace accepted by the model generating by COnfECt, varying the factors used.

Figure 3.17 shows the rate of valid traces accepted by COnfECt for $Conf8$ to $Cong10$, using both factors f_1, f_1' based on the identifier and f_2, f_2' based on the frequency of the events. We can also see in this figure that the models produced by the Strong strategy of COnfECt accept more valid traces than the models produced by the Weak strategy, which accept more valid traces than the models produced by the Strict strategy. However, this figure tends to show that the factors used do not affect the rate of valid traces accepted by the model. A manual analysis of the models reveals that the models generated with different factors are not identical, meaning that using different factors may lead to models with different precision.

3.6.5 RQ4: Rate of invalid traces accepted

Procedure: This time, this research question aims to measure the rate of invalid traces accepted by the model generated. The invalid traces are produced from the valid traces used in RQ3. The invalid traces are traces containing repetitions of actions, inversion of http request and response, or inversion of two events when a component calls an other one, i.e., when we have two consecutive sequences σ_1, σ_2 with σ_1 $weak\text{-}worr$ σ_2, the last event of σ_1 and the first event of σ_2 are inverted.

These invalid traces are then replayed on the model produced in RQ3, using every strategy of every method and kTail, in order to measure the rate of invalid traces accepted by each model. The factors used by COnfECt are the factors f_1, f_1'. A second experimentation is done on $Conf8$ to $Conf10$, with COnfECt, using the factors f_1, f_1' and f_2, f_2', in order to show the effect of the factors on the model precision.

Results: Figure 3.18 shows the rate of invalid traces that are accepted by the model generated by every method and strategy for each configuration. For COn-

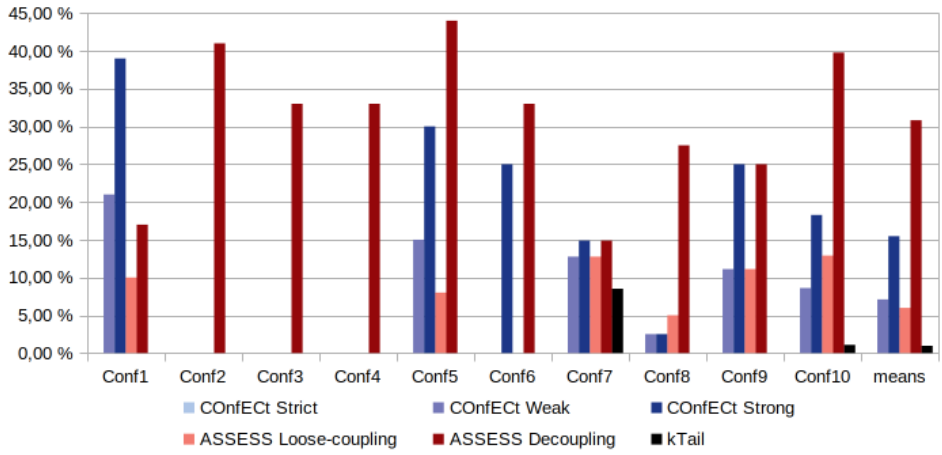


Figure 3.18: Rate of invalid traces accepted by the model generated by each method and strategy.

fECt, the models generated with the Strong strategy accept more invalid traces with an average of 15.47% than the ones generated with the Weak strategy with an average of 7.10% and the ones generated with the Strict strategy that accept none. It is not surprising that the Strict strategy, which is designed for not over-generalising the model, produces models that reject incorrect behaviors. However, the models generated by the two other strategies accept several invalid traces for some configurations. We have observed that in the traces given as input, the behavior of a component can be split, making that COnfECt may consider some single events as a single behavior, allowing the component to make an action multiple times or in inverted order in the model, such are designed our invalid traces. Both Weak and Strong strategies of COnfECt produce models that accept more invalid traces than the one produced by kTail, which accepts only an average of 0.96% of invalid traces. For ASSESS, the models obtained with Decoupled strategy accept an average of 31.96% of invalid traces, approximately one-third of the invalid traces. These invalid traces accepted are all traces where two events are inverted during the call of an other component. As the models produced with this strategy consider that the components are independent from each other and can act in parallel, these invalid traces are in this configuration of components valid, and so it is not surprising that they are accepted by the models. These traces represent approximately one-third of the invalid traces, explaining the results obtained with this strategy. The Loose-coupling strategy also accepts some invalid traces, with an average of 5.46% of accepted traces. This strategy also over-generalise the model during the last step of the method, where some synchronization actions can merge, producing new paths in the models that allow new behaviors. The only strategy that accepts fewer invalid traces than kTail is the Strict strategy of COnfECt; the other ones generate models that over-generalise the system and allow more behaviors than the

ones produced by the system.

In summary, ASSESS seems to produce models that accept more invalid traces than the models produced by COnfECt, that accept more invalid traces than the model generated by kTail.

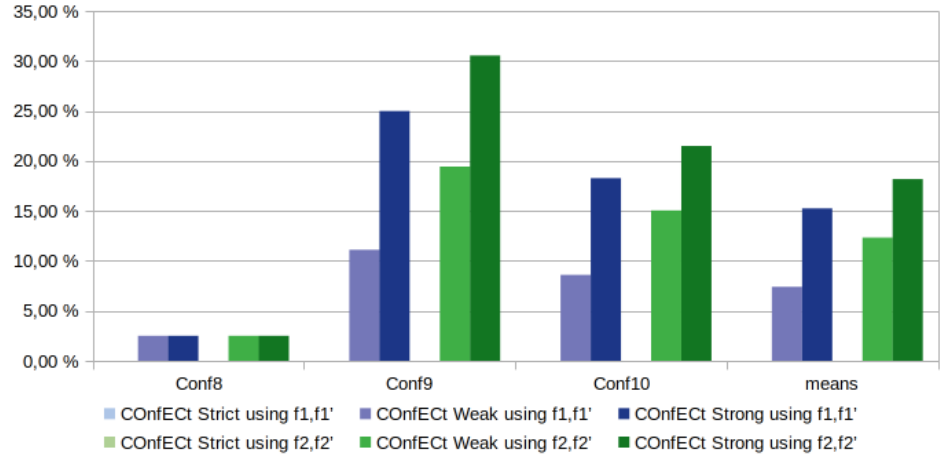


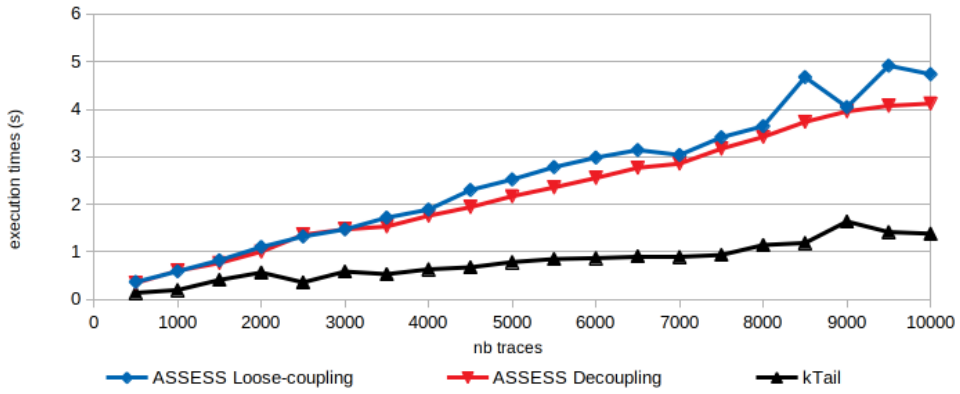
Figure 3.19: Rate of invalid trace accepted by the model generated by COnfECt, varying the factors used.

Figure 3.19 shows the rate of invalid traces accepted by COnfECt for $Conf8$ to $Cong10$, using both factors f_1, f_1' based on the identifier and f_2, f_2' based on the frequency of the events. Once again, we can see that the models generated with the Strong strategy accept more invalid traces than the ones generated with the Weak strategy, which accept more traces than the ones generated with the Strict strategy, which accept none. This figure tends to show that models generated using the factors f_1, f_1' based on the identifier are more precise than the ones generated with the factors f_2, f_2' based on the frequency of the events. Using frequency, like in f_2, f_2', does not guarantee that the separation of the events is perfectly done in the first step of COnfECt; the method may separate consecutive sequences of events that come from the same component. Even if these events are then grouped during the last step of the method, these separated sequences are then considered by the method as different behaviors that can be done separately by the system, over-generalising the models. However, when component identifiers are used, the method cannot separate consecutive sequences of events coming from the same component, as they will then have the same identifier.

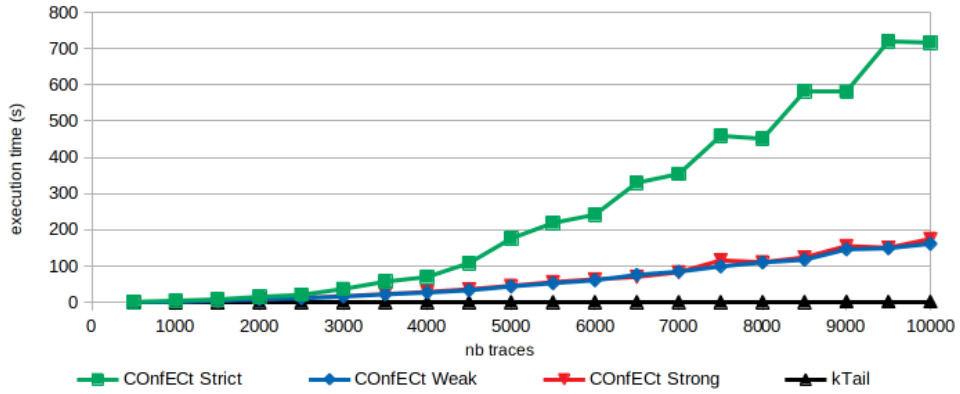
3.6.6 RQ5: Scalability

Procedure: To answer RQ5, we used our methods with both strategies on a set of traces extracted from the $Conf3$ and varying the number of traces given as inputs, from 500 to 10000 traces of 10 events, and measured the times taken to produce models. We also measured the times taken by kTail from the same logs. Different

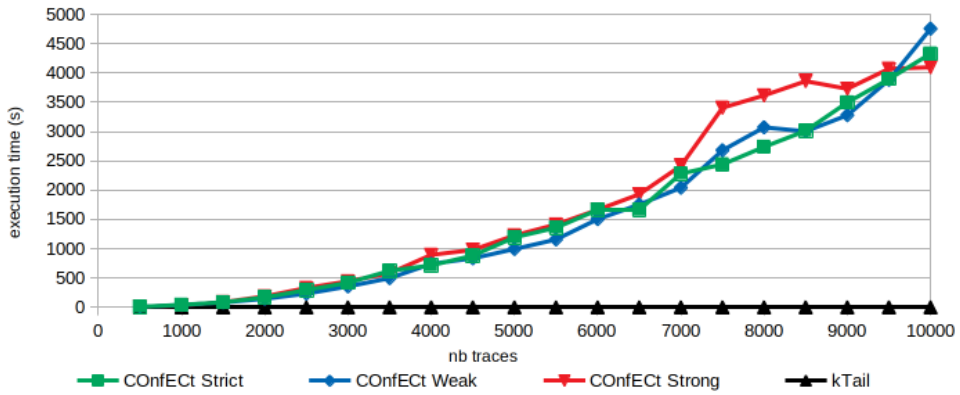
combinations of correlation and similarity factors are also used with COnfECt to illustrate the effects of these factors on execution times.



(a) Execution times of ASSESS.



(b) Execution times of COnfECt, using factors f_1, f_1'.



(c) Execution times of COnfECt, using factors f_2, f_2'.

Figure 3.20: Execution times of the different methods.

Results: Figure 3.20 shows the execution times of every method, according to the number of traces given as input. We can see that COnfECt and ASSESS are slower than kTail, no matter the strategy or factors used. It is not surprising. as COnfECt and ASSESS are based on a trace extraction algorithm to build one LTS per component, and they even call kTail in their last step. ASSESS is faster than COnfECt and never takes more than 5 seconds to produce models. Figure 3.20a shows that ASSESS tends to follow a linear time complexity for both strategies. More experiments reveal that even with 100,000 traces (one million events), the tool was still able to produce models in less than 2 hours. Figure 3.20b and 3.20c show the execution times of COnfECt, respectively, when the factors f_1, f_1' based on component identifiers are used and when the factors f_2, f_2' based on frequencies of events are used. These two figures reveal that COnfECt tends to follow a quadratic time complexity. COnfECt does not cover traces only once; it begins by separating traces into sub-sequences and then covers the sequences to split the traces, while ASSESS directly separates the events by covering the traces only once. We can see in Figure 3.20b that the Strict strategy is slower than the two other strategies when the factors f_1, f_1' are used. This difference stems from the multiple LTSs generation done by the Strict strategy. In Figure 3.20c we can see that COnfECt is a lot slower when the factors f_2, f_2' are used. These factors are more complex to evaluate than f_1, f_1', as they necessitate to cover traces to compute the frequencies of the events and their contents. With the factors f_2, f_2', the Strict strategy does not seem to be much slower than the two other strategies. observed that the Strict strategy with factors f_2, f_2' produces fewer LTSs than the Strict strategy with the factors f_1, f_1'. The difference in times is then more negligible, as LTSs generation times are reduced with the strict strategy, and the overall execution times are increased a lot during the analysis step.

In summary, ASSESS and COnfECt are both slower than kTail. COnfECt is always slower than ASSESS and is even slower if complex factors like f_2, f_2' are used. However, we believe that these tools can be used to produce models of systems producing a huge amount of events.

3.6.7 Threats to Validity

Some threats to validity can be identified in our evaluation. First, most of our test cases ($Conf1$ to $Cong6$) are similar and are built from the same type of component. Moreover, the implementations of both methods presented were done by different people and at different moments. This difference does not affect the precision of the model but may impact the efficiency of the implementations as none of them have been optimized, affecting the results obtained in RQ5.

COnfECt is more general and can be used on more systems, but it can be really difficult and necessitates a lot of knowledge on the system to define the correlation and similarity factors and determine the good thresholds that will lead to the inference of the good number of LTSs. This method is also limited by the hypothesis $H_{COnfECt}$, assuming that there is a single component that calls the others. This

hypothesis could be removed if we use the factor f_1, as it is possible to understand which component produced the event without assessing the others. However, as it cannot be removed for other factors, we choose to keep this hypothesis for COnfECt. On the other hand, ASSESS is a lot easier to use, as it does not necessitate to adjust thresholds. However, it can only be used if component identifiers are available in the traces. This last assumption is a major constraint, as we have observed that IoT systems do not always use such identifiers. The other assumptions also limit the generalization of the methods. It is common that the events are ordered by a timestamp, but not every system runs only one component at a time; complex IoT devices may run several components in parallel.

3.7 Summary

Two model learning methods for component-based IoT systems were presented in this chapter. Both have the same objective: to produce a system of LTSs representing the system, where each component is modeled by one LTS. COnfECt is more generally designed for component-based systems and is not specially designed for IoT. It can be used on many systems but necessitates an expert that has good knowledge of the system. ASSESS is designed for IoT systems, easier to use, but cannot be used on every IoT system as it necessitates component identifiers in the events in logs.

The results obtained with the experimentations tend to show that both methods produce models that are more readable than the ones produced by kTail in the case of a system composed of several components and comfort us that separating the components in different LTSs improves the readability of the models.

The communications between the components have to be hidden in the logs. This is a restrictive assumption that can be applied to component-based systems like, for example, complex embedded devices. However, this assumption makes it difficult to use the COnfECt and ASSESS on a system of IoT devices that communicate with each other. That is why another passive model learning method was next developed, designed for communicating systems. This method is described in the next chapter, and we go further in model learning by producing for each component a dependency graph representing the dependencies among the components of the system.

This work was published in several papers; the first four papers present the method COnfECt, and the last one presents the method ASSESS:

- Elliott Blot, Patrice Laurencot and Sébastien Salva. COnfECt : Une Méthode Pour Inférer Les Modèles De Composants D'un Système. In *17èmes journées AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, Grenoble, France, June 2018.

- Sébastien Salva and Elliott Blot. Confect: An approach to learn models of component-based systems. In *Proceedings of the 13th International Confer-*

ence on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018., pages 298–305, 2018.

- Sébastien Salva, Elliott Blot and Patrice Laurençot. 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*, pages 142–148, 2018.

- Sébastien Salva and Elliott Blot. Model generation of component-based systems. *Software Quality Journal*, 28(2):789–819, January 2020.

- Sébastien Salva and Elliott Blot. Reverse engineering behavioral models of iot devices. In *31st International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Lisbon, Portugal, July 2019.

Chapter 4

CkTail: Model learning of communicating systems

4.1 Introduction

IoT systems are generally composed of several devices like sensors, actuators, or gateways, called components in this chapter, which communicate with each other. We have shown in the previous chapter that the methods COnfECt and ASSESS cannot efficiently be used on such systems, as they both require that the interactions among the components are hidden. Active model learning methods [23, 47, 54] require that the components are testable, so controllable. We observed that it is difficult to apply on several communicating systems, like IoT systems that are made up of uncontrollable sensors. Passive model learning methods like CSight [9] or the method provided in [36] based on kbehavior tend to be good options as they take into account these communications among the components. However, we have observed that they suffer from one main issue related to the detection of sessions in logs. We call a session a temporary message interchange among components forming a behavior of the whole system from one of its initial states to one of its final states. Recognizing sessions in event logs helps extract "complete" traces and build more precise models.

In this chapter, we present the method Communicating kTail (CkTail), which produces one model per component, and tries to detect sessions in logs in order to build precise models. Moreover, the method goes further in model learning by generating for each component of the system a dependency graph that represents its interactions with the other components. The method takes as input a log extracted from the system and produces for each component of the system a model called Input Output Labeled Transitions System (IOLTS) representing its behavior and a Directed Acyclic Graph (DAG) representing its dependencies. The method is composed of four steps called *Log Formatting*, *Trace Extraction*, *Dependency Graph Generation*, and *IOLTS generation*. The first step aims to format a log into actions. The second one aims to analyze the actions and to extract traces represent-

ing sessions and component lists representing component dependencies. The third step produces dependency graphs. Finally, the last step uses the traces extracted in the second step in order to generate one behavioral model for each component.

The method has been implemented and experimented with with real use cases. The method is compared to several other model learning methods on several points: Rate of valid traces accepted by the models generated by the method, Rate of invalid traces accepted by the models generated by the method, and execution times. We also provide an evaluation of the dependency detection of our method. The evaluation shows that CkTail generates more precise models, detects the majority of the dependencies, and can generate models from large logs within reasonable time.

This chapter is structured as follows: First, in Section 4.2, the different models produced by CkTail are defined. An overview of the method is given in Section 4.3. Each step of CkTail is described in Section 4.4, and its evaluation is discussed in Section 4.5. Finally, we conclude with a summary of the chapter in Section 4.6.

4.2 Preliminary

In this chapter, the behavior of each component is expressed with an Input Output Labeled Transitions System (IOLTS). This model is an extension of the LTS model, such that the transitions are labeled by input or output actions. $\mathcal{L}$ denotes the set of actions expressing what happens in the system.

Definition 13 (IOLTS) *An Input Output Labeled Transition System (IOLTS) is a 4-tuple $\langle Q, q0, \Sigma, \rightarrow \rangle$ where:*

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

A finite sequence of actions in $\mathcal{L}^*$ is called a trace. When a sequence l is a subsequence of an other sequence l', we write $l \preceq l'$. We refer to the first element of a sequence l with $first(l)$ and to the last element of a sequence l with $last(l)$.

The dependency graphs of the components are expressed with Directed Acyclic Graphs representing the dependencies of the components.

Definition 14 (Directed Acyclic Graph) *A Directed Acyclic Graph (DAG) Dg is a 2-tuple $\langle V_{Dg}, E_{Dg} \rangle$ where V is the finite set of vertices and E the finite set of edges.*
λ denotes a labelling function mapping each vertice $v \in V$ to a label $\lambda(v)$

Each vertice represents a component of the system. If there exists in a dependency graph a path from a component c_1 to a component c_2, we say that c_1 depends on c_2.

4.3 Overview

CkTail aims to produce, for each component of the system under learning (SUL), an IOLTS representing its behavior and a dependency graph representing its dependencies with the other components of the system. The method takes as input an event log that can be collected with monitoring tools. This log is then analyzed to detect sessions, which will be used to produce IOLTSs.

Some assumptions are made on the system in order to efficiently detect sessions in logs:

- **A1 Event Log:** the communications among the components can be monitored on components, on servers, gateways, or by means of wireless sniffers. Event logs are collected in a synchronous environment made up of synchronous communications. Furthermore, the events have to include timestamps given by a global clock for ordering them. At the end of the monitoring process, we consider having one event log;
- **A2 Event content:** components produce events that include parameter assignments, allowing to identify components. We assume having in events two parameter assignments of the form $from := d$, $to := d$ expressing the source and the destination of an event. Other parameter assignments may be used to encode data. Besides, an event can be identified as a request or a response, or it can be an internal non-communication action.
- **A3 Device Collaboration:** components can run in parallel and communicate with each other. To learn precise models, we want to recognize sessions of the system in event logs. We assume that:
  - **A31:** The components of SUL follow this strict behavior: they cannot run multiple instances; requests are processed by a component on a first-come, first-served basis. Besides, every request is associated with at least one response. Or
  - **A32:** The events belonging to one session are identified by a session identifier.

The assumption A3 helps detect sessions in logs. The assumption A32 greatly eases the detection of sessions. However, we have observed that IoT devices seldom use session identifiers. That is why we also propose another analysis technique that necessitates the system to follow the assumption A31. We observed that this assumption can be applied to several IoT systems.

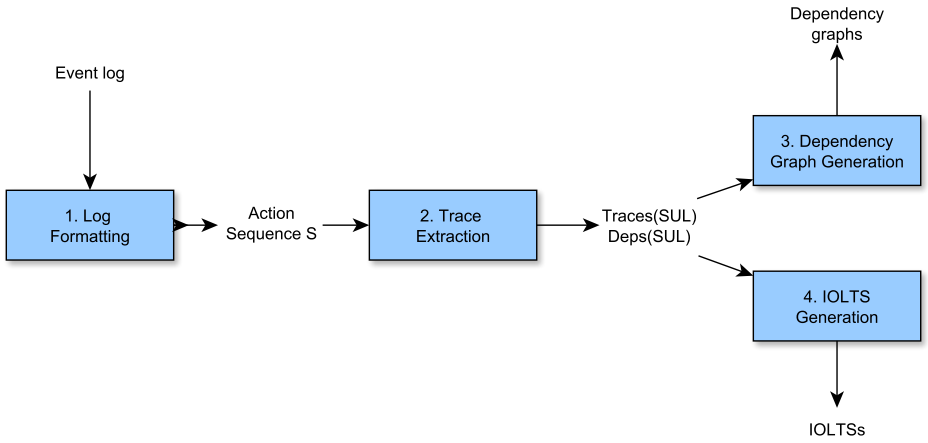


Figure 4.1: CkTail overview.

CkTail is composed of four steps depicted in Figure 4.1. The first step called *Log Formatting* formats raw events and returns a sequence S of actions of the form $a(\alpha)$ with a a label and α an assignment of parameters. The method uses some regular expressions, given by the user, that match the events and detect the label, the timestamps, and the different parameters of the events, as in the previous chapter.

Then, the second step *Trace Extraction* covers the action list and segments it into traces with the help of some constraints derived from the assumptions A1 to A3. These constraints express that a response is always in the same trace as its related request, nested requests (a request to a component that also performs another request before responding to it) belong to the same trace, actions performed by a component that already acted and performed in a limited time delay are kept in the same trace, and a sequence of actions sharing the same data are in the same trace. The resulting traces are gathered in the set $Traces(SUL)$. During this analysis, the method also tries to detect the dependencies among the components. These dependencies are collected under the form of component lists $c_1c_2 \dots c_k$ expressing that the component c_1 depends on a component c_2 that depends on another component and so on. These dependency lists are gathered in the set $Dep(SUL)$.

These dependencies are then used in the third step called *Dependency Graph Generation* to produce for each component of SUL a dependency graph. Finally, the last step, *IOLTS Generation*, produces the IOLTSs representing the behavior of the components and generalizes them with kTail. The CkTail steps are detailed in the next section.

4.4 Description of CkTail

4.4.1 Log Formatting

This first step aims to format the log given as input and returns an action sequence S, where the actions have the form $a(\alpha)$ with a a label and α an assignment of parameters in P, with P the set of parameter assignments. This step is the same as the first step of COnfECt and ASSESS described in the previous chapter.

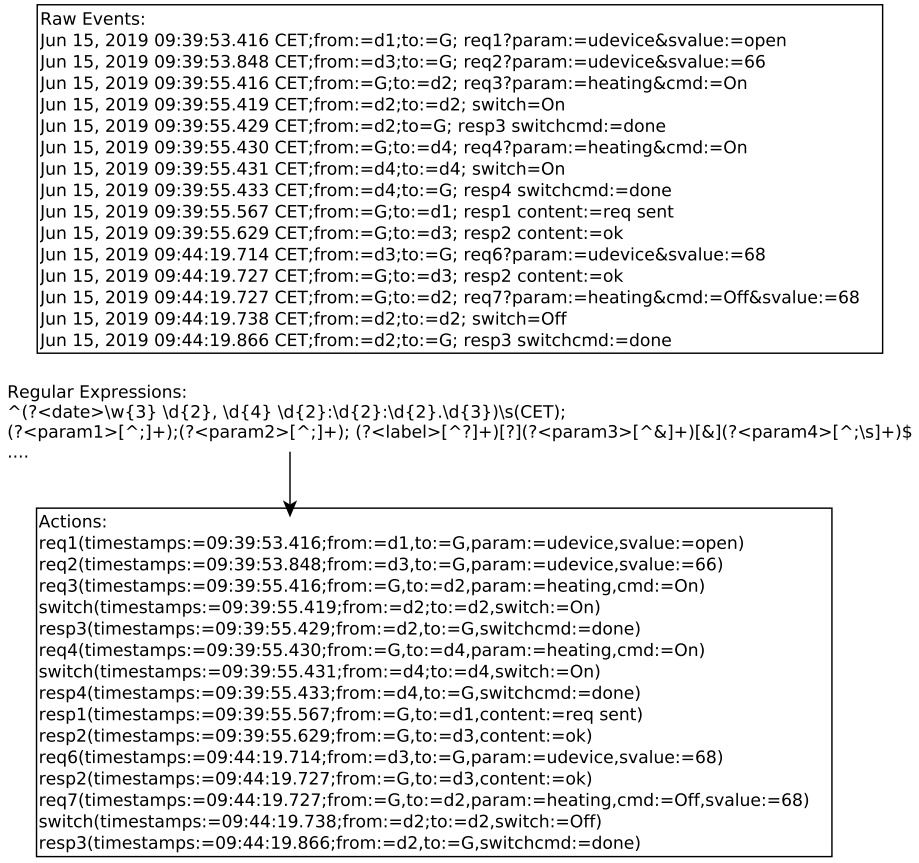


Figure 4.2: Example of Trace Formatting step.

Figure 4.2 shows an example of raw events and the corresponding actions.

The raw events contain timestamps, making them compliant with the assumption A1. For better readability in this example, the label of every action directly expresses if the action is a request, with a label starting with req, or a response with a label starting with $resp$. The other actions are non-communication actions. To format the actions, the method uses regular expressions that match the events and detect the timestamps, the label, and the different parameters. An example of regular expression is given in Figure 4.2. This expression matches the first raw

event of the figure. The group $< date >$ matches the timestamps of the event, the group $< label >$ matches the label $req1$, and finally, each group $< param >$ matches a parameter assignment in the events.

At the end of this step, we assume having a sequence $S \in \mathcal{L}^*$ of actions of the form $a_1(\alpha_1) \dots a_k(\alpha_k)$.

4.4.2 Trace Extraction

This step aims to detect sessions in the sequence of actions S and component dependencies. It returns a trace set $Traces(SUL)$, and a set of component lists $Dep(SUL)$ representing component dependencies.

Two algorithms are proposed in this section, respectively related to the assumption A31 when no session identifier is present in the events or A32 when identifiers are available in the events of the log.

Trace Extraction Without Session Identifiers (A31)

When no session identifier is present in the events, we suppose that SUL is compliant with A31, i.e., the components cannot run multiple instances at the same time, requests are processed on a first-come, first-served basis, and every request is associated with at least one response.

From the assumptions A1, A2, and A3, we extracted a set of constraints representing the conditions for a sub-sequence of S to be a session. The actions of the action sequence S are covered using these constraints to detect if they have to be added to current trace σ modeling a session. The constraints given in Table 4.1 allow to state whether an action of S belongs to σ. The first two constraints C1 and C2 express that a response has to be kept in the same session as its corresponding request. C3 expresses that nested requests have to be kept in the same session. The constraint C4 expresses that a component that has already participated in the session can send a new request in the same session if this request is sent shortly after other actions of the session, or the request shares data with previous actions of σ. Finally, the last constraint C5 concerns non-communication actions and expresses that a non-communication action stays in the session if it happens shortly after other actions of the session or shares data with previous actions of σ.

In order to formalize these constraints with boolean expressions and use them in our algorithm, the following notations are defined:

- $from(a(\alpha)) = c$ denotes the source of the action;
- $to(a(\alpha)) = c$ denotes the destination;
- $components(a(\alpha)) = \{from(a(\alpha)), to(a(\alpha))\}$;
- $time(a(\alpha)) = t$ returns the timestamps value identifying when $a(\alpha)$ occurred, $time(\epsilon) = +\infty$;

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

Table 4.1: Constraints derived from the assumptions A1, A2, A31. When one of these constraints holds, the current action $a_i(\alpha_i)$ of S is kept in a trace σ.

- $isReq(a(\alpha))$, $isResp(a(\alpha))$ are boolean expressions expressing if the action is a request or a response;
- $session(a(\alpha)) = id$ denotes the session identifier when available. Otherwise, $session(a(\alpha)) = \emptyset$.
- $data(a(\alpha)) = \alpha \setminus \{from := c_1; to := c_2, time := t, session := s\}$;
- KC stands for the set of known components involved in the session σ so far;
- $response(a_1(\alpha_1), a(\alpha))$ is the boolean expression $isResp(a_1(\alpha_1)) \wedge$ $from(a_1(\alpha_1)) = to(a(\alpha)) \wedge to(a_1(\alpha_1)) = from(a(\alpha))$;
- $Lreq(\sigma)$ denotes the set of sequences of pending requests i.e. the sequences of requests $a_1(\alpha_1) \ldots a_k(\alpha_k) \preceq \sigma$ for which responses have not yet been received. $LReq(\sigma) =_{def} \{a_1(\alpha_1) \ldots a_k(\alpha_k) \preceq \sigma \mid isReq(a_i(\alpha_i))_{1 \leq i \leq k}$, $\forall a(\alpha) \in \mathcal{L}^* : response(a(\alpha), a_i(\alpha_i)) \implies a_i(\alpha_i)a(\alpha)a_{i+1}(\alpha_{i+1}) \not\preceq \sigma\}$;
- $OLreq(\sigma)$ denotes the set of requests for which at least one response has been received;
- $ontime(a(\alpha), \sigma)$ is a boolean expression that returns true if the action $a(\alpha)$ may belong to the session σ with regard to the session duration or session time-out;
- *data-dependency*$(a(\alpha), S, \sigma)$ is a boolean expression that returns true if the request $a(\alpha)$ shares some data with other requests of the session $\sigma \preceq S$;

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

Table 4.2: Formalization of the constraints C1-C5 used in the trace extraction algorithm

- $pendingRequest(c)$ is the boolean expression ($\exists \sigma_1 \in Lreq(\sigma), a(\alpha) \in \sigma_1 : c \in components(a(\alpha))$) that evaluates whether the component c has sent (resp. received) a request and has not yet received (resp. sent) the response.

Table 4.2 gives the constraints of Table 4.1 expressed with these notations. Some of these notations are also used to formally define the notion of dependency. We consider that c_1 depends on c_2 when one of the following expressions holds:

Definition 15 (Component dependency) *Let $c_1, c_2 \in C$, $c_1 \neq c_2$, and $S \in \mathcal{L}^*$. We denote c_1 depends on c_2 iff $(c_1 \xrightarrow[\sigma]{r} c_2) \vee (c_1 \xrightarrow[\sigma]{nr} c_2) \vee (c_1 \xrightarrow[\sigma]{data} c_2)$ with:*

1. $c_1 \xrightarrow[\sigma]{r} c_2$ *iff* $\exists \sigma \preceq S, a(\alpha) \preceq \sigma : isReq(a(\alpha)), from(a(\alpha)) = c_1, to(a(\alpha)) = c_2$;
2. $c_1 \xrightarrow[\sigma]{nr} c_2$ *iff* $\exists \sigma \preceq S, a_1(\alpha_1) \dots a_k(\alpha_k) \preceq \sigma : from(a_1(\alpha_1)) = c_1, to(a_k(\alpha_k)) = c_2, a_1(\alpha_1) \dots a_k(\alpha_k) \in LReq(\sigma)$;
3. $c_1 \xrightarrow[\sigma]{data} c_2$ *iff* $\exists \sigma \preceq S, \alpha \in P : DS(\sigma, c_1, c_2, \alpha)$ *and* $\forall \sigma' = a'_1(\alpha'_1)a'_2(\alpha'_2) \dots a_k(\alpha_k) \preceq S : DS(\sigma', c_1, c_2, \alpha) \implies \sigma' \preceq \sigma$, *with* $DS(a_1(\alpha_1) \dots a_k(\alpha_k)c_1, c_2, \alpha)$ *the boolean expression* $from(a_1(\alpha_1)) = c_2 \wedge to(a_k(\alpha_k)) = c_1 \wedge isReq(a_k(\alpha_k)) \wedge to(a_i(\alpha_i)) = from(a_{i+1}(\alpha_{i+1}))_{1 \leq i < k)} \wedge \bigcap_{(1 \leq i \leq k)} \alpha_i = \alpha$.

The first expression expresses that c_1 depends on c_2 if c_1 sends a request to c_2. The second one expresses that c_1 depends on c_2 when c_1 queries c_2 with nested requests of the form $req1(from := c_1, to := c)req2(from := c, to := c_2)$. The last one refers to data dependency. We say that c_1 depends on c_2 if there is a unique

sequence of actions from c_2 ended by a request to c_1 sharing the same data α. This sequence has to be unique; otherwise, it becomes difficult to establish a correct dependency as several components are potential candidates. In order to not deduce false dependencies among components, no dependency is kept if several sequences of actions, all sharing the same data and addressed to several components, are found.

Algorithm 8: Trace Extraction with A31

input : Action sequence S
output: $Traces(\text{SUL})$, Component set C, Component dependency set $Deps(\text{SUL})$

```
1  C := Deps(SUL) := ∅;
2  Keep-or-Split(S);
3  Procedure Keep-or-Split(a_1(α_1) ... a_k(α_k)) is
4      σ := σ_2 := ϵ;
5      Lreq(σ) := OLreq(σ) := ∅;
6      KC := components(a_1(α_1));
7      i := 1;
8      while i ≤ k do
9          updateOLreq(a_i(α_i));
10         case C1 true do
11             σ := σ.a_i(α_i); Trim(σ_r);
12             KC := KC ∪components(a_i(α_i));
13         case C1 false and C2 true do
14             σ := σ.a_i(α_i);
15             KC := KC ∪components(a_i(α_i));
16         case C3 true do
17             σ := σ.a_i(α_i);
18             Extend(σ_r, a_i(α_i));
19             KC := KC ∪components(a_i(α_i));
20         case C3 false and C4 true do
21             σ := σ.a_i(α_i);
22             Extend(ϵ, a_i(α_i));
23             KC := KC ∪components(a_i(α_i));
24         case C5 true do
25             σ := σ.a_i(α_i);
26             KC := KC ∪components(a_i(α_i));
27         otherwise do σ_2 := σ_2.a_i(α_i) ;
28         i++;
29     Traces(SUL) := Traces(SUL) ∪ {σ};
30     C := C ∪ KC;
31     if σ_2 ≠ ϵ then
32         Keep-or-Split(σ_2);
33 END;
```

Algorithm 8 describes the Trace Extraction algorithm when no session identifiers are available. This algorithm takes as input the action sequence S and returns the set of traces $Traces(\text{SUL})$, the set of components C and the set of dependencies $Deps(\text{SUL})$. The procedure ***Keep-or-Split*** covers the action sequence $a_1(\alpha_1) \dots a_k(\alpha_k)$ given as input and tries to extract one trace σ representing a session. First, the procedure adds the components that participate in the first action of the sequence in KC, the set of known components (line 6). Then the procedure covers every action of the trace to determine if it has to be kept in the trace σ. For every action $a_i(\alpha_i)$ of the sequence, the set $OLreq(\sigma)$ is updated with the help of the procedure $updateOLreq$ given in Algorithm 9 (line 9). More precisely, if $a_i(\alpha_i)$ is a new request coming from a component c, then all the previous requests that involve c are removed to meet A31 (first come, first served). In the same way, if $a_i(\alpha_i)$ is a response, only the request associated with this response is kept. If $a_i(\alpha_i)$ is a request coming from a component c, all the previous requests from c are removed from $OLreq(\sigma)$ in order to follow the first come, first served basis of A31. Otherwise, if $a_i(\alpha_i)$ is a response coming from a component c, only the request from c associated with this response is kept. Then, ***Keep-or-Split*** checks if one of the constraints C1-C5 holds with the action $a_i(\alpha_i)$. If it is the case, $a_i(\alpha_i)$ is added to the trace σ, and the set KC is updated according to the components found in this action. If it is not the case, the action is added to the sequence σ_2. If the constraint C1 holds for the action $a_i(\alpha_i)$, the procedure $Trim$ given in Algorithm 9 is called (line 11), in order to remove the last request of σ_r and to put it in the set $OLreq(\sigma)$ instead, as $a_i(\alpha_i)$ is the response associated to this request. If the constraint C3 or C4 holds for the action $a_i(\alpha_i)$, the procedure $Extend$ is called in order to update the set $Lreq(\sigma)$ and the dependencies (C3 implies that the second point of Definition 15 holds (nested requests), and C4 may imply that the third point holds (data dependency)). Once the sequence is covered by ***Keep-or-Split***, the trace σ is added to $Traces(SUL)$, and the component set C is updated with KC (line 29-30). The procedure ***Keep-or-Split(σ_2)*** is then recursively called if σ_2 is not empty, in order to extract other traces (line 32).

The expression $ontime$ is implemented with the procedure $ontime$. Several implementations are possible; one of them is given in Algorithm 9. This implementation checks if the delay between the action $a_i(\alpha_i)$ and the beginning of the session represented by σ is lower than a duration, denoted T in the procedure. The implementation of the procedure ***data-dependency*** used in C4 and C5 is also given in Algorithm 9. It checks if there is a data dependency between $a_i(\alpha_i)$ and the actions of σ, as defined by the third point of Definition 15.

Figure 4.3 shows an example of results obtained with this algorithm. For better readability, the parameters of the actions of $Trace(SUL)$ are hidden. The trace in red in the figure is the first one that was built by the algorithm. The first action $req1$ was put in a first trace, and the dependency $d1G$ is added in the set $Dep(SUL)$ as this first action is a request (first point of Definition 15). The other actions are then covered by the algorithm and analyzed in order to decide if they are kept in the same trace with respect to the constraints $C1$ to $C5$. The algorithm determines

Algorithm 9: Procedure used by Trace Extraction step.

Procedure $updateOLreq(a_i(\alpha_i))$ **is**
  if $isReq(a_i(\alpha_i))$ **then**
    $OLreq(\sigma) := OLreq(\sigma) \setminus \{a(\alpha) \in OLreq \mid from(a_i(\alpha_i) \in components(a(\alpha))\}$;
  else if $isResp(a_i(\alpha_i))$ **then**
    $Lr := \{a(\alpha) \in OLreq(\sigma) \mid from(a_i(\alpha_i) = to(a(\alpha))\}$
    $OLreq(\sigma) := OLreq(\sigma) \setminus \{a(\alpha) \in OLreq(\sigma) \mid from(a_i(\alpha_i) \in components(a(\alpha))\} \cup Lr$;
Procedure $Trim(\sigma_r)$ **is**
  $\sigma' := remove(last(\sigma_r))$;
  $Lreq(\sigma) := Lreq(\sigma) \setminus \{\sigma_r\} \cup \{\sigma'\}$;
  $OLreq(\sigma) := OLreq(\sigma) \setminus \{a(\alpha) \in OLreq(\sigma) \mid from(last(\sigma_r)) \in components(a(\alpha))\}$;
  $OLreq(\sigma) := OLreq(\sigma) \cup \{last(\sigma_r)\}$;
Procedure $Extend(\sigma_r, a(\alpha))$ **is**
  $\sigma' := \sigma_r.a(\alpha) = a_1(\alpha_1) \dots a_k(\alpha_k)$;
  $Lreq(\sigma) := Lreq(\sigma) \setminus \{\sigma_r\} \cup \{\sigma'\}$;
  //Component dependencies
  $lc := c_1 \dots c_k c_{k+1}$ such that $c_i = from(a_i(\alpha_i))_{(1 \le i \le k)}$, $c_{k+1} = to(a_k(\alpha_k))$;
  $Deps(\text{SUL}) := Deps(\text{SUL}) \cup \{lc\}$;
Procedure $ontime(a_i(\alpha_i), \sigma)$ **is**
  return $(time(a_i(\alpha_i)) - time(first(\sigma)) < T)$;
Procedure *data-dependency*$(a_i(\alpha_i), S, \sigma)$ **is**
  if $\exists \sigma_1 = a_1(\alpha_1)a_2(\alpha_2) \dots a_i(\alpha_i) \preceq S : to(a_i(\alpha_i)) \xrightarrow[\sigma_1]{data} from(a_1(\alpha_1))$ **then**
    $Deps(\text{SUL}) := Deps(\text{SUL}) \cup \{to(a_i(\alpha_i)).from(a_1(\alpha_1))\}$;
    if $\sigma_1 \preceq \sigma.a_i(\alpha_i))$ **then**
      return true;
  return false;

that no constraint holds for the second action $req2$, meaning that this action is not added in the trace in red. The next request $req3$ is added to the trace as it forms with $req1$ nested requests (C3). Consequently, the dependency $d1Gd2$ is also added to the set of dependencies (second point of Definition 15). As $req3$ is a request from G to $d2$, the dependency $Gd2$ is also added to $Dep(SUL)$. The following non-communication action $switch$ is also added to the trace as it was done shortly after the other actions of the session (C5) by a component that previously participated in the session. The response $resp3$ is added to the trace as it is the response of the request $req3$ already in the trace (C1). The following request $req4$ forms other nested requests with the request $req1$. Consequently, the dependencies $d1Gd4$

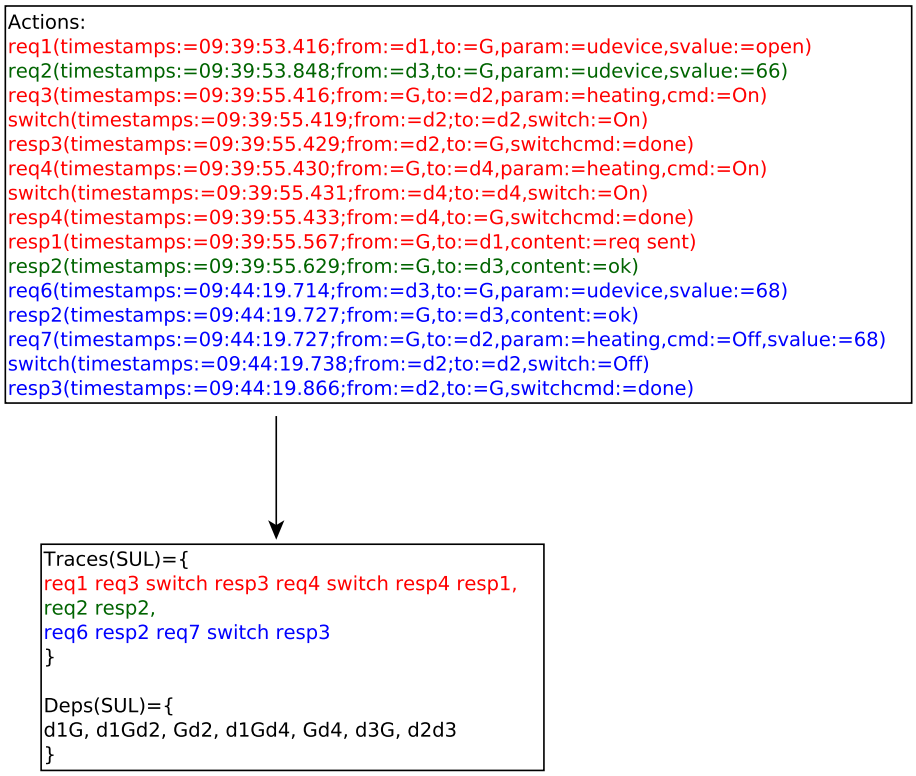


Figure 4.3: Example of trace extraction with Algorithm 8.

and $Gd4$ are added to the set of dependencies. Finally, the next two responses, $resp4$ and $resp1$, are added as they are the corresponding responses for $req4$ and $req1$ (C1). No constraint holds for the remaining actions. Then, the algorithm processes the remaining actions and builds two more traces. The actions in blue are separated from the ones in green, as too much time elapsed between them (C4). The request $req2$ in green gives the dependency $d3G$ shown in the set $Dep(SUL)$. In the trace in blue, the requests $req6$ and $req7$ are kept in the same trace, as the sub-sequence $req6$ $req7$ follows the condition of a data dependency by sharing the data $svalue := 68$. This data dependency gives the dependency list $d2d3$ stored in $Deps(SUL)$.

Trace Extraction With Session Identifiers (A32)

We now suppose that session identifiers are available in the events. The Trace Extraction algorithm can be simplified as the identifiers already denote in which session the actions belong. Algorithm 10 shows this new algorithm.

Since session identifiers are available in actions, fewer efforts are needed to detect sessions and extract traces. However, this step still aims to detect the depen-

Algorithm 10: Trace Extraction with A32

input : Action sequence S
output: $Traces(\text{SUL})$, Component set C, Component dependency set $Deps(\text{SUL})$

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

dencies among the components of SUL. Algorithm 10 begins by generating the traces of $Traces(SUL)$ such that all the actions of a trace share the same identifier (line 3) . Then, the traces are covered in order to produce the set $Dep(SUL)$, using the constraints C1, C3, and C4 of Table 4.2. If C3 holds, this means that the algorithm detects nested requests (second point of Definition 15). With the constraint C4, the algorithm detects data dependencies (third point of Definition 15). The constraint C1 is only used by the algorithm to update the set of pending requests $Lreq$.

Figure 4.4 shows an example of a result obtained with the step Trace Extraction. Each color represents a trace built by the algorithm using session identifiers assigned to the parameter idx. The dependency set $Dep(SUL)$ is then built by covering the traces and by checking if the conditions C3 or C4 hold. The dependencies $d1G, Gd2, Gd4, d3G$ are added to the set $Dep(SUL)$ as the algorithm detects requests where these components act. The dependencies $d1Gd2$ and $d1Gd4$ are added as nested requests are found with C3. Finally, the dependency $d2d3$ is added as a data dependency is found between these two components via the requests $req6$ and $req7$ which share the same data $svalue := 68$.

The two algorithms presented in this section can return slightly different re-

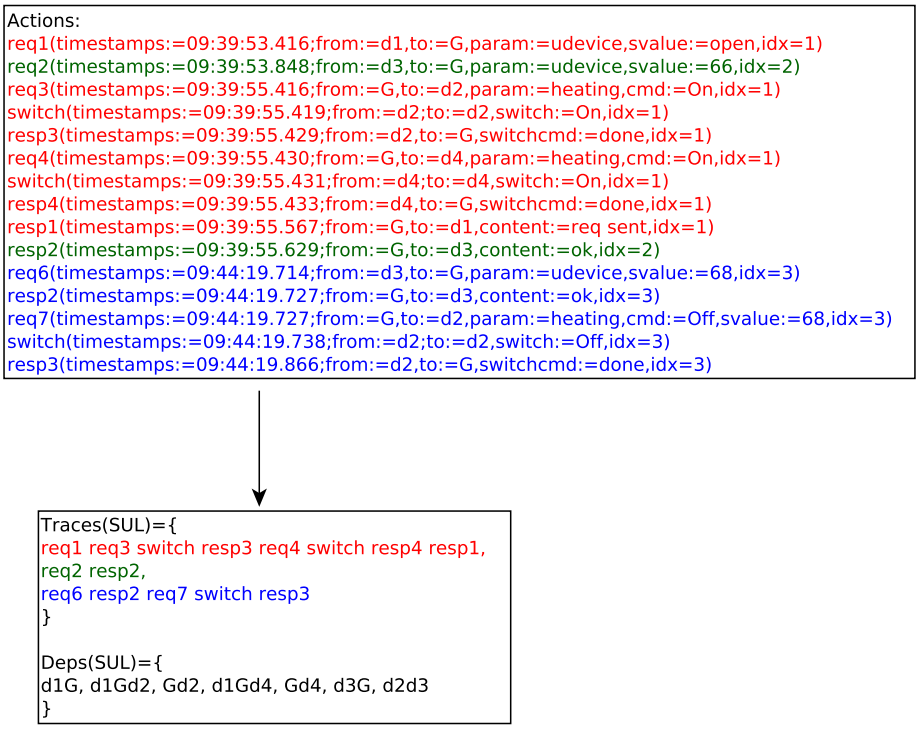


Figure 4.4: Example of result obtained with Algorithm 10.

sults of dependencies. Some differences can be found concerning the detection of data dependencies (third point of Definition 15). In Algorithm 8, when no session identifier is present in the actions, the detection of action sequence having data dependencies is done on the complete action sequence S, while Algorithm 10 does it on every trace. As the traces are shorter than the complete action sequence, the dependencies are more precise if Algorithm 10 is used.

4.4.3 Dependency Graph Generation

This step produces for each component a dependency graph representing its dependencies. Algorithm 11 implements the Dependency Graph Generation step. It takes as input $Dep(SUL)$ and groups the lists that start by the same component, with the equivalence relation $\sim_c$ on C^* given by $\forall l_1, l_2 \in Deps(\text{SUL})$, with $l_1 = c_1 \dots c_k$, $l_2 = c'_1 \dots c'_k$, $l_1 \sim_c l_2$ iff $c_1 = c'_1$. Each partition will be then used to build a DAG, where each state represents a component. The paths of the DAGs are derived from the component lists. Finally, the algorithm computes the transitive closure of the DAGs to make all the component dependencies visible.

Figure 4.5 shows an example of dependency graph generated by Algorithm 11. Each dependency list of the set $Dep(SUL)$ has given a path in a DAG. Each DAG

Algorithm 11: Device Dependency Graph Generation

input : $Deps(\text{SUL})$
output: Dependency graph set DG
1 **foreach** $C_i \in Deps(\text{SUL})/\sim_c$ **do**
2 **foreach** $c_1 c_2 \dots c_k \in C_i$ **do**
3 add the path $s_{c_1} \rightarrow s_{c_2} \dots s_{c_{k-1}} \rightarrow s_{c_k}$ to Dg_i;
4 Dg'_i is the transitive closure of Dg_i;
5 $DG := DG \cup \{Dg'_i\}$;

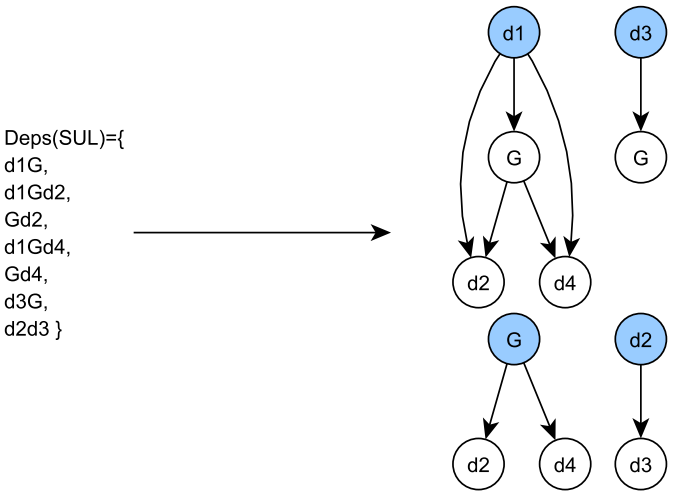


Figure 4.5: Example of dependency graph generated by the step Dependency Graph Generation.

represents the dependencies of one component (in blue in the figure). The transitive closure of the DAGs is finally computed.

4.4.4 IOLTS Generation

The last step of CkTail generates one IOLTS for every component of C. It is implemented by Algorithm 12, which takes as input the set $Trace(SUL)$, to produce for each component c an IOLTS L_c representing its behavior.

The algorithm starts by transforming the traces to integrate the notions of input and output. Given a trace $a_1(\alpha_1) \dots a_k(\alpha_k)$, every action is doubled by separating the component source and destination (line 4-7). The source and the destination are identified by a new assignment on the parameter id added to each action. Besides, the parameter assignments are generalized by removing timestamps and session identifiers. For a communication action $a_i(\alpha_i)$, this step produces a new trace σ' composed of the output $!a_i(\alpha_{i1})$ sent by the source of the action, followed by the

Algorithm 12: IOLTS Generation

input : $Traces(\text{SUL})$
output: IOLTSs $L_{c_1} \dots L_{c_k}$

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

input $?a_i(\alpha_{i2})$ received by the destination (lines 5-7). Non-communication actions are not doubled in the new traces and considered as output actions ($!a_i(\alpha_{i1})$. Then, the trace σ' is segmented into sub-sequences, each capturing the behaviors of one component only (lines 8- 9). The trace set T_c gathers the traces of the component c. Finally, each set T_c is used to produce an IOLTS L_c by producing for each sub-sequence $\sigma = a_1(\alpha_1) \dots a_k(\alpha_k) \in T_c$ a path starting and ending to the initial state. $q0 \xrightarrow{\{a_1(\alpha_1)\}} q_1 \dots q_{k-1} \xrightarrow{\{a_k(\alpha_k)\}} q0$.

Definition 16 (IOLTS generation) *Let $T_c = \{t_1, \dots, t_n\}$ be a trace set. $L_c = \langle Q, q0, \Sigma, \rightarrow \rangle$ is the IOLTS derived from T_c where:*

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

$$\frac{t_i = a_1(\alpha_1) \dots a_k(\alpha_k)}{q0 \xrightarrow{\{a_1(\alpha_1)\}} q_{i1} \dots q_{ik-1} \xrightarrow{\{a_k(\alpha_k)\}} q0}$$

Then, Algorithm 12 applies the kTail algorithm to generalize the IOLTSs by merging the equivalent states having the same $kfuture$. We used $k = 2$, as recommended in [33, 31].

Figure 4.6 shows an example of IOLTS generated by the last step of CkTail. Three traces were extracted from the log of Figure 4.3 after the Trace Extraction step. In Figure 4.6, the trace $req2(from := d3, to := G) resp2(from := G, to := d3)$ in green in the figure is transformed into $!req2(from := d3, to := G, id := d3)$ $?req2(from := d3, to := G, id := G)$ $!resp2(from := G, to := d3, id := G)$ $?resp2(from := G, to := d3, id := d3)$. Two traces are extracted:

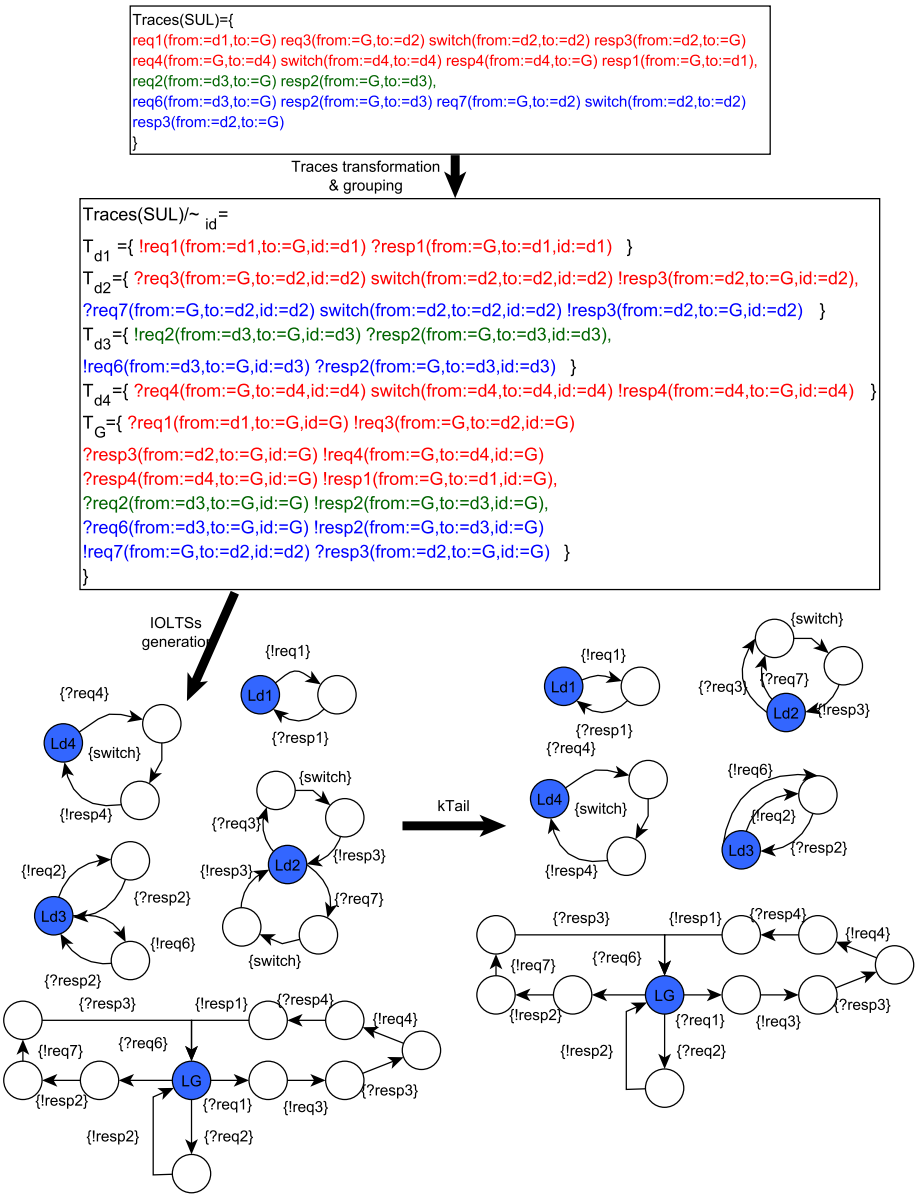


Figure 4.6: Example of IOLTS generated by CkTail.

$!req2(from := d3, to := G, id := d3)$ $?resp2(from := G, to := d3, id := d3)$, and the trace $?req2(from := d3, to := G, id := G)$ $!resp2(from := G, to := d3, id := G)$, each representing some behaviors of G and $d3$. The algorithm has inferred 5 trace sets as SUL is made up of 5 components. The IOLTSs $Ld1$ and $Ld3$ are reduced, as they are the only ones that had state with the same $2future$.

4.5 Evaluation

The method was implemented in Java and is available as open source software[1]. This implementation takes as input a log and regular expressions, and produces IOTLSs and dependency graphs stored in DOT files. We used this implementation on several real-life use cases in order to answer the following research questions:

- RQ1 (Rate of valid traces accepted): is CkTail able to produce models that accept valid behaviors of SUL?
- RQ2 (Rate of invalid traces accepted): is CkTail able to produce models that reject invalid behaviors of SUL?
- RQ3 (Dependency Detection): is CkTail able to recover accurate dependencies among components?
- RQ4 (Scalability): how does CkTail scale with the size of the event log?

The first two research questions target the precision of the models produced by CkTail by first assessing its capacity to build models that accept valid traces of the system in RQ1 and its capacity to build models that reject invalid traces of the system in RQ2. A valid trace is here trace extracted from the system that was not used for the generation. An invalid trace is a trace that expresses an incorrect behavior of the system. We will compare CkTail with the model learning methods CSight[9], ASSESS presented in Chapter 3 using both strategies, and the method denoted LFkbehaviour, proposed in [36]. The third research question targets the capacity of CkTail to detect accurate dependencies among components. As no other model learning method produces dependency graphs, the results will be compared with the ones obtained from a manual analysis of the dependencies observed in the systems. The last research question targets the performance and scalability of CkTail. Execution times of CkTail will also be compared with the ones of the model learning methods CSight, ASSESS, and LFkbehaviour.

4.5.1 Setup

We constructed 6 configurations from a set of 7 devices: 3 sensors, 2 gateways, and 2 actuators. The number of devices and their interactions vary from one configuration to one another. At least one gateway was used for each configuration; the sensors communicate with gateways, and the latter send orders to actuators, according to the data received. These configurations are denoted $Conf1$ to $Conf6$. From each configuration, we extracted a log of approximately 2200 actions. Two other systems were also considered. The first one, denoted $Conf7$ is composed of 8 sensors that send data to a gateway. A log of 2206 events was extracted from this system. The second one $Conf8$ is a camera connected to NTP, SMTP, and

[1] https://github.com/Elblot/CkTailv2

FTP servers, and that can be queried by a user. A log of 1310 events was extracted from this system. All these logs do not include session identifiers. Hence, we manually modified them to compare our algorithms. The modifications consisted of manually adding session identifiers in actions with regard to the functioning of the systems. We denote these new logs $Conf9$ to $Conf16$. We denote CkTail-w/oS the method CkTail using Algorithm8 (when no session identifier is present in the events), and CkTail-w/S the method CkTail using Algorithm 10 (when session identifiers are present in the events).

As CSight was unable to produce a model for any of these configurations, even after 5 hours of execution, we chose to add two trace sets that were given with CSight, denoted Tcp and $AltBit$. Tcp contains 8 traces (46 events) collected from two components exchanging TCP events. $AltBit$ contains 15 traces (246 events) expressing exchanges between two components using the Alternating Bit Protocol. These traces do not contain session identifiers.

4.5.2 RQ1: Rate of valid traces accepted

Procedure: This research question studies the precision of the models produced by CkTail by assessing the rate of valid traces accepted by them. i.e., its capacity to produce models that accept valid behaviors of SUL. In order to answer this question, we separated the log of every configuration with a ratio of approximately 70-30%. The first 70% of the log were used to generate models using CkTail-w/S, CkTail-w/oS, CSight, ASSESS with the Decoupling and Loose-coupling strategy, and LFkbehaviour. The last 30% were used to produce valid traces. These valid traces were produced manually to avoid splitting sessions. We obtained between 35 and 202 traces for $Conf1$ to $Conf16$. As we have trace sets and not logs for Tcp and $AltBit$, the sets were split with the same approximate ratio of 70-30%, with again the first 70% used for the generation and the last 30% as valid traces. As the sets of traces are very small for these two use cases, we obtained 2 valid traces for Tcp and 4 for $AltBit$.

Results: Figure 4.7 illustrates the percentages of valid traces accepted by the models generated by the different tools for every configuration. CSight could only produce models of Tcp and $AltBit$ and did not succeed in producing any model for any other configuration, even after 5 hours of computation. However, these configurations are very small, and all the methods succeeded in producing models that accept all valid traces, making it difficult to compare CSight and the other methods. CkTail is the method that accepts the most traces, with an average of 96.03% of valid traces accepted for CkTail-w/oS and 96.97% for CkTail-w/S. As expected, models generated by CkTail-w/S are slightly more precise than those generated by CkTail-w/oS, as the session detection is more precise. The close results of the two methods tend to show that Algorithm 8 is efficient for detecting sessions, though. The models generated by ASSESS accept fewer traces than the ones generated by CkTail but more than the ones generated by LFkbehaviour. The

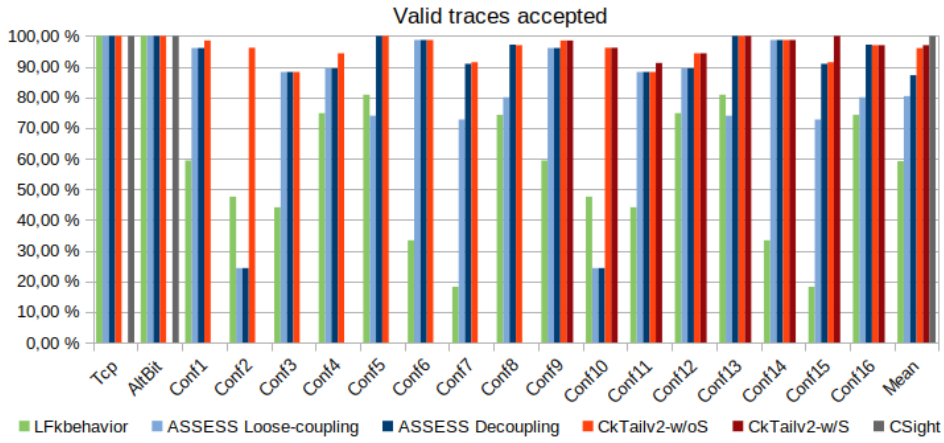


Figure 4.7: Percentage of valid traces accepted by the models with every configuration.

models generated by ASSESS with the Loose-coupling strategy accept an average of 80.37% and 87.19% with the Decoupling strategy, while the models generated by LFkbehaviour accept an average of 59.16%. ASSESS with the decoupling strategy inferred models that accept slightly more valid traces for $Conf8$ and $Conf16$, with 97.14% of the valid traces accepted, where both models produced by CkTail accept 96.97% of them.

4.5.3 RQ2: Rate of invalid traces accepted

Procedure: This research question also targets the precision of CkTail by assessing the rate of invalid traces accepted by them. In order to answer this question, we produced invalid traces from the set of valid ones generated in RQ1 and replayed them on the models generated in RQ1. These invalid traces were produced by applying one of the following mutations to the valid ones: repetitions of requests, repetitions of responses, repetitions of non-communication actions, or inversion of a request and its associated responses. A set of 43 to 100 invalid traces was generated for $Conf1$ to $Conf16$, and 20 for Tcp and $AltBit$.

Results: The diagram of Figure 4.8 shows the proportions of invalid traces accepted by the models generated by the methods for every configuration. This figure shows that the models generated by CkTail-w/S and CkTail-w/oS methods reject all the incorrect behaviors. As for the previous research question, CSight can only be compared to the other methods via the configurations Tcp and $AltBit$. Both models generated by CkTail and CSight reject all the incorrect behaviors for these two configurations, making it again difficult to compare CSight with CkTail. We can see in the figure that the models generated by ASSESS and LFkbehaviour ac-

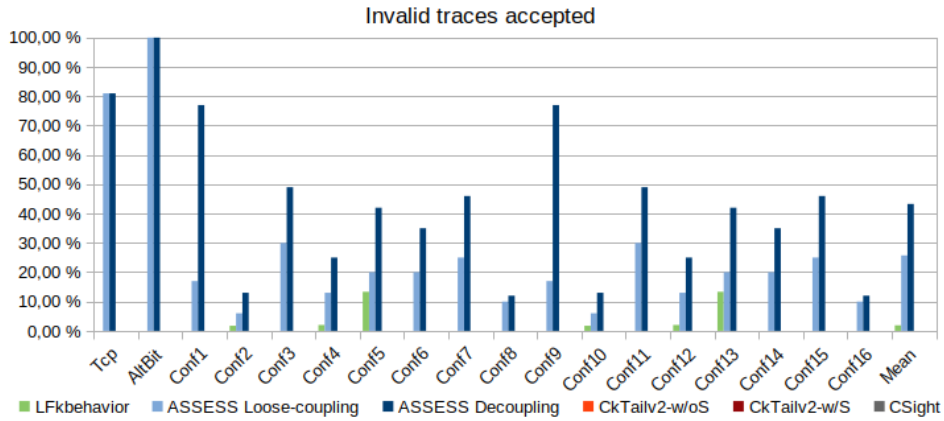


Figure 4.8: Percentage of invalid traces accepted by the models for each configuration.

cept some invalid traces. In the case of models generated by ASSESS, a lot of invalid traces are accepted, with an average of 43.28% of the invalid traces accepted with the Decoupling strategy and 25.72% with the Loose-coupling strategy. A manual analysis of the models reveals that some actions are missing in some models. ASSESS is not designed to produce models of a system where the communications among components are visible. These incomplete models accept some actions repetitively or in disorder, making the model accept some invalid traces. The models generated by LFkbehaviour accept an average of 1.90% of the invalid traces. We observed that in some models, a component can send a response before receiving a request, making it accept traces with repetitive responses or with requests and responses inverted.

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

4.5.4 RQ3: Dependency Detection

Procedure: This research question targets the capacity of CkTail to recover correct dependencies among components. In order to answer this question, we generated the dependency graphs of the components of SUL using both methods CkTail-w/oS and CkTail-w/S, and we compared the results we obtained with the dependencies we recovered from a manual analysis. This experiment was performed on $Conf1$ to $Conf8$ for CkTail-w/oS, and $Conf9$ to $Conf16$ for CkTail-w/S. Even if $Conf9$ to $Conf16$ also follow the assumption A31, we prefer not to use CkTail-w/oS on these configurations as the parameters representing the session identifiers may falsify the results by adding false data dependencies. For each of these config-

urations, we evaluated the recall of the dependency detection, i.e., the rate of real dependencies found by the methods, and the precision of the dependency detection, i.e., the rate of dependencies found by the method that are real.

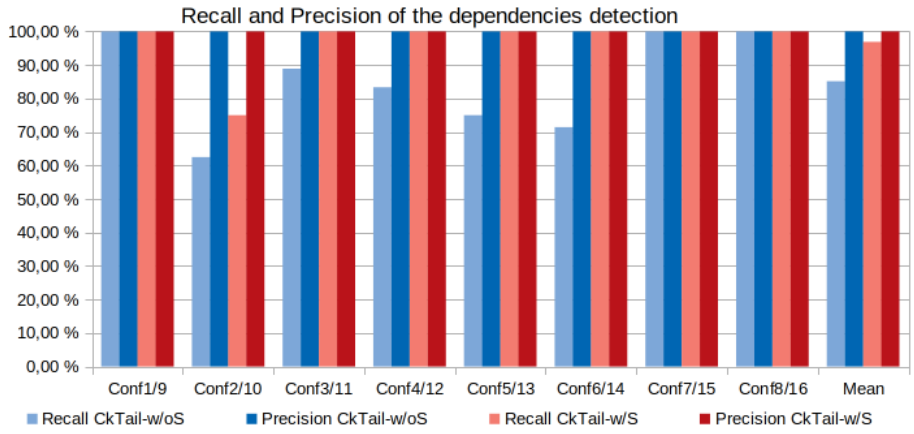


Figure 4.9: Recall and precision of CkTail to detect component dependencies. Recall is the percentage of the real dependencies that are detected; precision is the percentage of detected dependencies that are correct.

Results: Figure 4.9 shows the recall and precision of the dependencies detection of CkTail on every configuration. On average, CkTail-w/oS detected 85.14% of the real dependencies and CkTail-w/S 96.88%. The missing dependencies are all data dependencies. We have observed that in some cases, several sequences of actions sharing the same data are detected at the same time, leading to several potential dependencies on the same component. In the third point of Definition 15, we choose to not consider them in order to avoid creating false dependencies. This case appears more often with CkTail-w/oS than with CkTail-w/S, as the last one looks for data dependencies on the traces instead of the complete action sequence. This difference in data dependencies search explains the difference in valid traces accepted between both versions. No wrong component dependency was returned by both algorithms, showing that we succeeded in not returning false dependency in the system.

These results tend to show that CkTail detects most of the dependencies and does not detect the wrong dependencies.

4.5.5 RQ4: Scalability

Procedure: This research question targets the capacity of our method to produce models using large event logs. In order to answer this question, we used the methods on $Conf3$, collecting several event logs that contain from 500 to 10,000 events,

and we measured execution times. As stated earlier, CSight was unable to produce models for this system. In order to compare CSight with the other methods, we also measured the times spent to produce the models of Tcp and $AltBit$. These experiments were done with a computer with 1 Intel® CPU i5-6500 @ 3.2GHz and 32GB RAM.

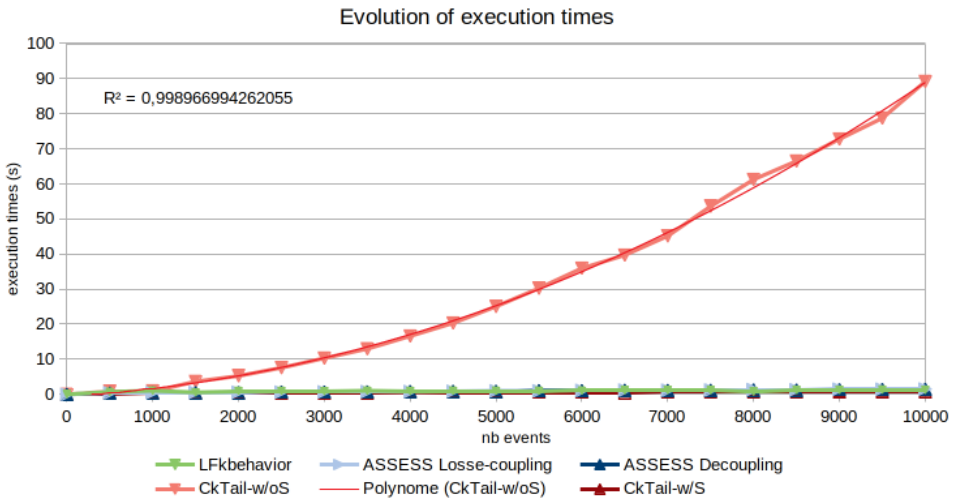


Figure 4.10: Execution times vs. number of events

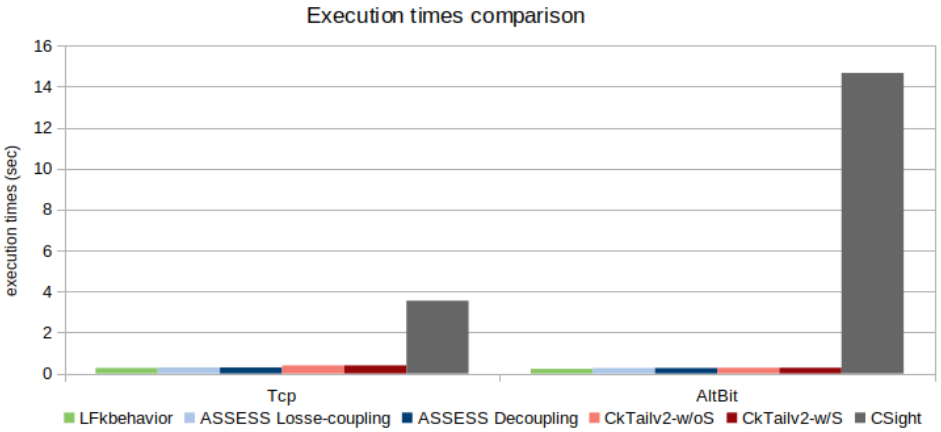


Figure 4.11: Execution times of the tools with the configurations Tcp ans $AltBit$

Results Figure 4.10 shows the evolution of execution time of CkTail, ASSESS, and LFkbehaviour, according to the size of the log. As we can see, CkTail-w/oS gives longer execution times than the other method. This difference is due to the

session identification step that is not done in ASSESS and LFkbehaviour and which is greatly simplified in CkTail-w/S. As we can see in the figure, the time complexity of CkTail-w/oS is quadratic, while the time complexities of CkTail-w/S, ASSESS, and LFkbehaviour are linear. For a log containing 10000 events, CkTail-w/oS took approximately 90 seconds, where the other methods took all less than 2 seconds (1.26 seconds with LFkbehaviour, 0.8 seconds with CkTail-w/S, and 1.5 seconds with ASSESS using both strategies).

Figure 4.11 shows the execution times of every method on Tcp and $AltBit$. These results tend to show that CSight is significantly slower than the other tools. CSight took respectively 3.5 and 14.6 seconds to produce the models of Tcp and $AltBit$, while the other methods took less than 0.4 seconds for each configuration. As stated previously, CSight could not produce models of the other configurations even after 5 hours of computation.

These experiments show that CkTail can be used in practice to infer models of communicating systems even with large event logs, but it suffers from insufficient scalability on account of its feature of detecting sessions for extracting traces in the case where no session identifier is present in events.

4.5.6 Threats to Validity

Some threats to validity can be identified in this evaluation. First, most of our use cases are IoT systems where the components communicate only with the HTTP protocol. We have also considered some configurations that use a different protocol with the configurations Tcp and $AltBit$, but these cases are too small to give any conclusion. This is a threat to validity because many communicating systems rely on other kinds of protocols, from which it may be more difficult to identify senders, receivers, requests, or responses. Hence, our results cannot be generalized. This is why we deliberately avoid drawing any general conclusion. We chose to mainly concentrate our experiments on IoT systems that we devised to be able to appraise the capability of CkTail of inferring correct dependency graphs. This threat is somewhat mitigated by the fact that our results can be easily generalized to communicating systems based upon the HTTP protocol and that the latter is used by numerous communicating systems.

In order to correctly detect sessions, Algorithm 8 and Algorithm 10 use a procedure $ontime$, which uses a threshold that can vary according to SUL. It can be difficult to find the correct threshold that will lead to a precise model. Figure 4.12 shows the evolution of the precision of the models generated from $Conf3$ by varying the threshold used in the procedure $ontime$ given in Algorithm 9. As we can see, four different threshold values can be considered as good values. When the threshold value is difficult to set, we suggest producing several models using different thresholds in order to choose the most appropriate one.

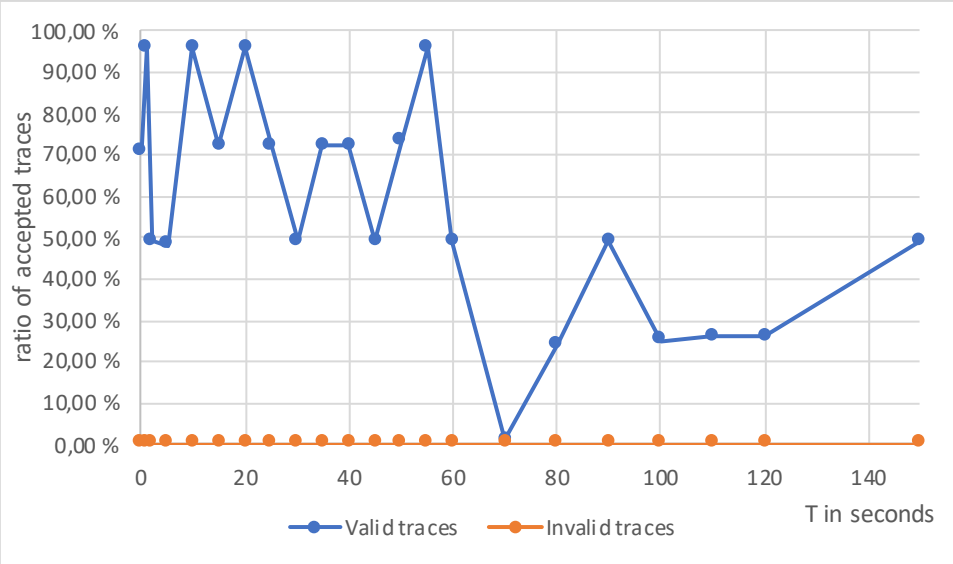


Figure 4.12: Precision of the models generated by CkTail according to the threshold used by the procedure $ontime$.

4.6 Summary

In this chapter, a new passive model learning method for communicating systems called CkTail was presented. This method aims to produce, from an event log, the models of a system composed of devices or components that communicate with one another. For each component of the system, CkTail produces one IOLTS representing its behaviors and a dependency graph representing its dependencies with the other components. The method also aims to produce more precise models by trying to detect sessions in logs. CkTail-w/S can be used when session identifiers are present in the events in logs; otherwise, CkTail-w/oS can be used.

The evaluation of CkTail shows that it produces more precise models than the other methods designed for communicating systems, except for CSight, as we do not succeed in efficiently comparing both approaches. However, we have determined that our method can be used on larger logs than CSight, making it more usable on real systems. The results also show that CkTail succeeds in finding the majority of the component dependencies and is more precise in this detection in the case where session identifiers are present in the events.

The models generated can greatly help analyze the components of the system. Such models can be used to assess the security of a device with the help of formal methods, for example. In the next chapter, we will present the method SMProVer, which helps an auditor assess the security of an IoT system from the models generated by CkTail.

This work was published in a paper:

- Sébastien Salva and Elliott 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, pages 27–38.

Chapter 5

SMProVer: Verification of Security Measures Implementation

5.1 Introduction

In the previous chapter, we have presented several methods that produce models that describe the behavior of the different components of the system. Now that the models of the components are produced, we want to analyze them in order to detect in their behavior some security flaws. In methods using models to assess the security of a system, the models are generally produced by hand by an expert [51, 12, 35]. These models are specific to the method used to assess them, like in [41], meaning that it is difficult to use the models produced automatically by CkTail with these methods. As the models produced by CkTail describe the behavior of the components, we can assess the different components of the IoT system separately. A system is as weak as its weaker component; assessing each component separately and securing the weaker ones will lead to a more secure system.

In order to assess the models produced by CkTail, we present in this chapter the method SMProVer (Security Measures Properties Verification) that aims to automatically verify the models produced. This method combines model learning and model checking to verify that some security measures are implemented by every component of the system. Generally, when we want to verify some properties on a model, we have to write the properties using the alphabet used by the model. This task can be long and difficult, and the properties need to be rewritten if we want to use them in a different model. The goal of our method is to transform the models and a set of existing generic properties with the help of an expert system in such a way that the properties can be verified on the models. The generic properties could then be used on several systems and do not need to be rewritten for every system we want to verify. The method is composed of four steps, called *Model Learning*, *Model Completion*, *Properties Instantiation*, and *Properties Verification*. In

the first step, the models of the components are generated with the help of CkTail. The models are then complete in the second step, which adds new propositions in the labels of transitions, which will be used during verification. In the third step, the generic properties are modified according to the data that can be found in the models in order to make the verification of the properties on the models possible. Finally, the last step calls a model checker to verify the properties on the models.

Some experiments done with the help of a tools chain are also provided in this chapter, where we show the precision of the method to find that security measures are implemented and that issues are in the system. We also provide an experiment showing the time taken by our tools chain to produce the models, the properties, to verify them, and to show how the execution time scales with the size of the inputs.

This chapter is structured as follows: First, some preliminary definitions are given in Section 5.2. Then an overview of the method is given in Section 5.3. The method is described in detail in Section 5.4, and Section 5.5 provides an evaluation done by our implementation. Section 5.6 gives some limitations of the method, and finally Section 5.7 summarizes the chapter.

5.2 Preliminary

The goal of the method is to verify that every component of an IoT system implements security measures. For that, we will make verifiable a set of predefined properties, representing the security measures, on models describing the behavior of the components of the system. To do that, two things are needed: the behavior of the components that will be mined from a log of the system and represented in models, and the properties representing the security measures.

As in the previous chapter, the behavior of a component is modeled by an IOLTS (Input Output Labeled Transition System). First, CkTail will only produce IOLTSs where each transition has a unique label representing the action done in the system. This action can be a message from a component to another or an internal non-communicating action. For SMProVer, we need that the transitions of the IOLTSs are labeled by a set of labels. This transformation can easily be done by transforming each label of transition representing the action into a set of labels containing only the label representing the action. In the following, we consider that CkTail produces such models, where the transitions are labeled by a set of labels containing only one label. The method SMProVer will then complete this set of labels in order to make the properties verifiable on the models.

The models will be assessed to determine if some security recommendations are implemented by the components. We can find such lists of security measures for IoT systems in the literature [60, 29, 45]. Among them, the ENISA organization provides several documents, listing security measures linked to IoT gathered from different sources, according to different contexts, like, for example, smart cars [19], smart manufacturing [18], or in the context of critical information infrastructures [17]. We choose to focus on security measures in the context of critical

information infrastructures [17], as the security measures provided in this paper are the most general and can be applied to more IoT systems. This document proposes 57 security technical measures, denoted GP-TM-01 to GP-TM-57, that should be implemented by each device of the system. We have determined that the implementation of at least 11 of them can be observed on the IOLTSs, representing the behavior of the components. These security measures cover a wide range of security considerations, such as security by design, data protection, risk analysis, etc.

These security measures are formalized in this chapter with the help of LTL properties. An LTL property is constructed by using the following grammar: $\phi ::= p \mid (\phi) \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi\mathbf{U}\phi$, with AP a set of atomic propositions and $p \in AP$. Additionally, a LTL property can be constructed with the following operators, each of which is defined in terms of the previous ones: $\top \stackrel{def}{=} p \vee \neg p$, $\bot \stackrel{def}{=} \neg\top$, $\phi_1 \wedge \phi_2 \stackrel{def}{=} \neg(\neg\phi 1 \vee \neg\phi_2)$, $\mathbf{F}\phi \stackrel{def}{=} \top\mathbf{U}\phi$, $\mathbf{G}\phi \stackrel{def}{=} \neg\mathbf{F}(\neg\phi)$. We choose to express aspects of the security measures with LTL properties because they are simpler to write than CTL properties [57]. Moreover, we can find in the literature patterns that help the user to produce the wanted properties [16]. The method described in this chapter could also work for CTL properties if LTL properties are not suited for representing a security measure. The LTL properties given as inputs to our methods are called *property types*. A property type is a general LTL property that captures the temporal relationships between predicates.

Definition 17 (Property type)

- *$Pred$ denotes a set of predicates of the form p (nullary predicate) or $p(x)$ with x a variable that belongs to the set X;*
- *The domain of a predicate variable $x \in X$ is denoted $Dom(x)$;*
- *A property type Φ is a LTL property built up from predicates in $Pred$. $\mathscr{P}$ denotes the set of property types.*

A property type is composed of predicates that should be instantiated before being verified on the models of the components; a property type cannot be verified on the models as it is. The predicates used in our property types used in this thesis are listed in Table 5.1. All the unary predicates have variables that can be assigned to values in $C \cup P$ with C the component set of SUL and P the parameter assignments given in the actions in the labels of the models.

With the help of these predicates, we produced 11 property types, each representing an aspect of a security measure extracted from [17]. These property types are listed below:

GP-TM-18: *Ensure that the device software/firmware, its configuration, and its applications have the ability to update Over-The-Air (OTA), that the update server is secure, that the update file is transmitted via a secure connection, that it does not*

Predicate	**Short Description**
$begin$	beginning of a new session
end	end of a session
$from(c)$	the message came from c
$to(c)$	the message is sent to c
$Request$	the message is a request
$Response$	the message is a response
$input$	the message is an input
$output$	the message is an output
getUpdate(x)	message including an update file
cmdSearch-Update	the component received the order to search for an update
$sensitive(x)$	the data x includes sensitive data
$credential(x)$	the data x includes credentials
$encrypted(x)$	the data x is encrypted
$searchUpdate$	the component requests for an update
$loginAttempt(c)$	authentication attempt with c
$authenticated(c)$	successful authentication with c
$loginFail(c)$	failed authentication with c
$lockout(c)$	c is locked due to repetitive authentication failures
$password\text{- }Recovery$	component uses a password recovery mechanism
$blackListed\text{-}Word$	message includes black listed words
$validResponse$	correct response with positive status
$errorResponse$	response containing an error message
$Unavailable$	the component that received the request is unavailable
$XSS(x)$	data x includes an XSS attack
$SQLinjection(x)$	data x includes an SQL injection attack

Table 5.1: Predicates defined from 11 ENISA measures related to communications.

contain sensitive data (e.g., hardcoded credentials), that it is signed by an authorized trust entity and encrypted using accepted encryption methods, and that the update package has its digital signature, signing certificate, and signing certificate chain, verified by the device before the update process begins.
This property checks that the component updates from an authenticated entity, and that the update file is encrypted and does not contain sensitive data. It checks that the proposition $getUpdate(f)$ can be true in the model, with f representing the update file, and that if the proposition $getUpdate(f)$ is true, the proposition $encrypted(f)$ is also true (the file f is encrypted), and the proposition $sensitive(f)$ is false (the file f does not contain sensitive data). Moreover, the proposition $getUpdate(f)$ cannot be true between the beginning of a trace ($begin$), and the authentication of the sender of the update file ($authenticated(c)$).

$$\mathcal{F}\, getUpdate(f) \wedge (\mathcal{G}(getUpdate(f) \rightarrow (encrypted(f) \wedge \neg sensitive(f)))) \wedge \mathcal{G}((begin \wedge \mathcal{F}\, end) \rightarrow (\neg(getUpdate(f) \wedge from(c))\, \mathcal{U}(authenticated(comp) \vee end)))$$

GP-TM-19: *Offer an automatic firmware update mechanism.*
The goal of this property is to check that the component can automatically search for updates. It checks that the proposition $searchUpdate$ is true in the model, and if it is the case even if the proposition $cmdSearchUpdate$ was never true (the component never receives the command to search a new update).

$$\mathcal{F}\, searchUpdate \wedge (\mathcal{F}\, searchUpdate \rightarrow \neg(\mathcal{F}\, searchUpdate \rightarrow (\neg searchUpdate\, \mathcal{U}(input \wedge cmdSearchUpdate \wedge \neg searchUpdate))))$$

GP-TM-24: *Authentication credentials shall be salted, hashed, and/or encrypted*, and **GP-TM-40:** *Ensure credentials are not exposed in internal or external network traffic*.
This property checks that credentials are encrypted during authentication. It checks that if the proposition $(loginAttempt(comp) \wedge credential(x))$ holds (authentication phase with credentials x, then the proposition $encrypted(x)$ is also true (credentials x are encrypted).

$$\mathcal{G}((loginAttempt(comp) \wedge credential(x)) \rightarrow encrypted(x))$$

GP-TM-25: *Protect against 'brute force' and/or other abusive login attempts. This protection should also consider keys stored in devices.*
This property checks that the account is locked after five authentication attempts in a row. It checks that between the proposition $begin$ and end (during only one run), if there is five times in row an authentication ($authenticated(c)$) that leads to the failure of the authentication ($loginFail(c)$), then the proposition $lockout(c)$ has to be true (c is locked).

$$\mathcal{G}((begin \wedge \mathcal{F}\, end) \rightarrow (\neg(\mathcal{G}((begin \wedge \mathcal{F}(end \vee authenticated(c)) \rightarrow P(5, c))) \rightarrow (\neg end\, \mathcal{U}\, lockout(c)))\, \mathcal{U}\, end)$$
$$\text{with } P(n, c) = ((\neg(loginFail(c)) \wedge \neg(end \vee authenticated(c)))\, \mathcal{U}(end \vee authenticated(c) \vee ((loginFail(c) \wedge \neg(end \vee authenticated(c)))\, \mathcal{U}(end \vee authenticated(c) \vee P(n-1, c))))) \text{ for } n > 0,$$
$$\text{and } P(0, c) = (\neg(loginFail(c))\, \mathcal{U}(end \vee authenticated(c)))$$

GP-TM-26: *Ensure password recovery or reset mechanism is robust and does not supply an attacker with information indicating a valid account. The same applies to key update and recovery mechanisms.*
This property checks that the password recovery system does not show too much information. It checks that if the proposition $passwordRecovery$ is true, then the proposition $blackListedWord(x)$ is false (x does not contain forbidden information).

$$\mathcal{G}(passwordRecovery \rightarrow \neg blackListedWord(x))$$

GP-TM-38: *Guarantee the different security aspects confidentiality (privacy), integrity, availability, and authenticity of the information in transit on the networks or stored in the IoT application or in the Cloud.*
This property checks that sensitive data are encrypted. It checks that if the proposition $sensitive(x)$ is true (x contains sensitive data), then $encrypted(x)$ is also true (x is encrypted).

$$\mathcal{G}(sensitive(x) \rightarrow encrypted(x))$$

GP-TM-42: *Do not trust data received and always verify any interconnections. Discover, identify and verify/authenticate the devices connected to the network before trust can be established, and preserve their integrity for reliable solutions and services.*
This property checks that the component needs authentication before sending or receiving data. It checks that before the component sends data for an other purpose that an authentication ($(Request \wedge to(c) \wedge \neg loginAttempt(c))$, or sends a positive response for an other purpose that an authentication ($(validResponse \wedge to(c) \wedge \neg loginAttempt(c))$ the component c is authenticated ($authenticated(c)$).

$$\mathcal{G}((\neg(validResponse \wedge to(c) \wedge \neg loginAttempt(c))\,\mathcal{U}(authenticated(c))) \wedge (\neg(Request \wedge to(c) \wedge \neg loginAttempt(c))\,\mathcal{U}(authenticated(c))))$$

GP-TM-48: *Protocols should be designed to ensure that, if a single device is compromised, it does not affect the whole set.*
This property checks that if an other component is unavailable, this component stays available. It checks that if a dependency d is unavailable ($(from(d) \wedge Unavailable)$), the component modeled continues to send $output$ messages (it does not become unavailable).

$$\neg\,\mathcal{G}(from(d) \wedge Unavailable) \rightarrow \neg(\neg output\,\mathcal{U}(output \wedge Unavailable)$$

GP-TM-52: *Ensure web interfaces fully encrypt the user session, from the device to the backend services, and that they are not susceptible to XSS, CSRF, SQL injection, etc.*
Two properties are written from this technical measure:

- The first property checks that the device returns an error if it received a request containing XSS. It checks that if the component receives a request containing XSS attack from a component $c1$ ($Request \wedge from(c1) \wedge to(c2) \wedge XSS(x)$), the component does not send a valid response to $c1$ until it has sent an error response to $c1$.

$$\mathcal{G}((Request \wedge from(c1) \wedge to(c2) \wedge XSS(x)) \rightarrow ((\neg(Response \wedge to(c1) \wedge from(c2)))\mathcal{U}((Response \wedge to(c1) \wedge from(c2)) \wedge (errorResponse \vee (\neg validResponse \wedge Response)))))$$

- The second property checks that the device returns an error if it received a request containing SQL injection. It checks that if the component receives a request containing SQL injection from a component $c1$ ($Request \wedge from(c1) \wedge to(c2) \wedge SQLinjection(x)$), the component does not send a valid response to $c1$ until it has sent an error response to $c1$.

$$\mathcal{G}((Request \wedge from(c1) \wedge to(c2) \wedge SQLinjection(x)) \rightarrow ((\neg(Response \wedge to(c1) \wedge from(c2)))\mathcal{U}((Response \wedge to(c1) \wedge from(c2)) \wedge (errorResponse \vee (\neg validResponse \wedge Response)))))$$

GP-TM-53: *Avoid security issues when designing error messages.*
ss This property checks that error messages sent in the network do not contain too much information. It checks that if the proposition $(errorResponse \vee \neg validResponse)$ is true (a response with an invalid status), then the proposition $blackListedWord(x)$ is false (x does not contain forbidden information).

$$\mathcal{G}((errorResponse \vee \neg validResponse) \rightarrow \neg blackListedWord(x))$$

The LTL properties were written by composing the LTL patterns given in [16]. These patterns help to structure LTL properties with precise and correct statements that model common situations, e.g., the absence of events or cause-effect relationships.

5.3 Overview

The goal of SMProVer is to help an auditor verify some generic LTL properties on every component of the system. The method takes as inputs a log of the communications between the components and a set of property types $\mathscr{P}$ representing the security measures we want to verify.

Figure 5.1 shows an overview of the method. The method can be separated into four steps called *Model Learning*, *Model Completion*, *Properties Instantiation*, and *Properties Verification*. First, during the step *Model Learning*, the method uses the log of communications collected from the system, with the help of regular expressions matching the events of the log, to produce for each component c_i, an IOLTS $\mathscr{L}(c_i)$ expressing its behavior and a dependency graph $Dg(c_i)$ representing its interactions with the other components of the system. The models of the components are then completed in the second step, *Model Completion*, in order to inject the propositions used in the property types in the labels of the IOLTSs. This injection of labels is done automatically with the help of an expert system that uses transformation rules on the set of transitions of the models. During this step, a new

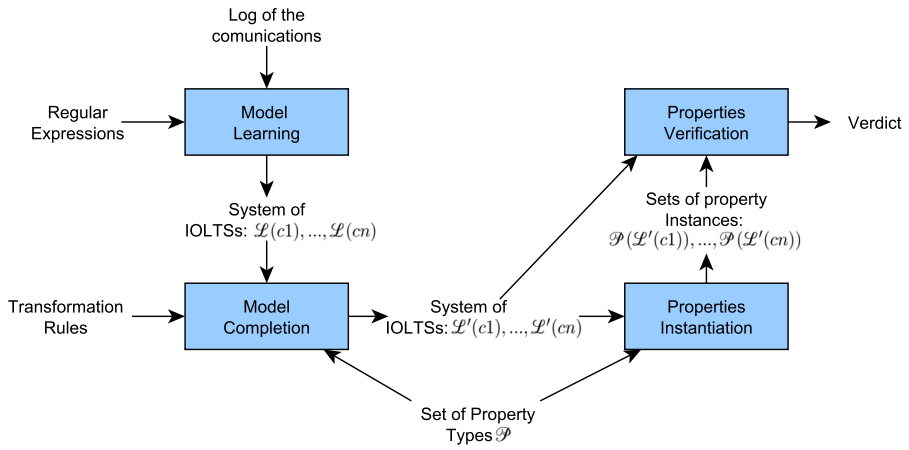


Figure 5.1: Overview of SMProVer.

IOLTS $\mathscr{L}'(c_i)$ is produced for each component c_i, where the set of labels of each transition of $\mathscr{L}(c_i)$ is improved with propositions of Table 5.1. These new models have labels that contain propositions instantiated according to the data that can be found in the model. In the third step of the method *Properties Instantiation*, the propositions of the property types are transformed for each model of component $\mathscr{L}'(c_i)$, according to the different values that can be found in the model. Each model has its own set of instantiated properties $\mathscr{P}(\mathscr{L}'(c_i))$ that can then be verified on it. This verification is done in the fourth and final step of SMProVer *Properties Verification*, which calls a model checker that verifies on each model $\mathscr{L}'(c_i)$, its associated set of properties $\mathscr{P}(\mathscr{L}'(c_i))$, and returns for each component a verdict.

Each of these steps is described with more details in the next section and is illustrated with examples.

5.4 SMProVer

5.4.1 Model Learning

The first step of the method aims to produce the models of the components of the IoT system. For that, we have already presented in the previous chapter, the method CkTail that was developed for this purpose. As the log taken as input by SMProVer needs to also be taken as input by CkTail, the log has to follow the assumptions expressed in the previous chapter, and those are recalled below:

- **A1 Event Log:** the communications among the components can be monitored on components, on servers, gateways, or by means of wireless sniffers. Event logs are collected in a synchronous environment made up of synchronous communications. Furthermore, the events have to include times-

tamps given by a global clock for ordering them. At the end of the monitoring process, we consider having one event log;

- **A2 Event content:** components produce events that include parameter assignments, allowing to identify components. We assume having in events two parameter assignments of the form $from := d$, $to := d$ expressing the source and the destination of an event. Other parameter assignments may be used to encode data. Besides, an event can be identified as a request or a response, or it can be an internal non-communicating action.
- **A3 Device collaboration:** components can run in parallel and communicate with each other. To learn precise models, we want to recognize sessions of the system in event logs. We assume that:
  - **A31:** The components of SUL follow this strict behavior: they cannot run multiple instances; requests are processed by a component on a first-come, first-served basis. Besides, every request is associated with at least one response. Or
  - **A32:** The events belonging to one session are identified by a session's identifier.

CkTail takes as input the log of the communications between the components and regular expressions that match these events and returns for each component an IOLTS representing its behavior and a dependency graph representing its interactions with the other components of the system. Figure 5.2 shows an example of a result we can obtain with CkTail. This method works in four steps. First, in the step *Log Formatting*, the log is formatted, with the help of the regular expressions, into a sequence of events of the form $a(\alpha)$ with a a label and α some parameter assignments. We can find in the action sequence of Figure 5.2 two parameters $from$ and to in α that respectively refer to the source and the destination of the events. In the second step *Trace Extraction*, the sequence of events is then analyzed in order to detect the dependencies and split the different sessions in the log, *i.e.* the behaviors that start from the initial state and that end to a final state of the system. We have shown in the previous chapter that session's detection leads to a more precise model. Each session detected by the method is put in a trace. Then in the third step of the method *Dependency Graphs Generation*, the dependencies gathered during the previous step are used to produce for each component its dependency graph. Finally, the last step, *IOLTSs Generation*, uses the traces of the session extracted during the second step to produce the behavioral models of the components. The events of the traces are first split into sub-sequences in such a way that every sub-sequence contains only events related to only one component. These sub-sequences are then used to produce for each component an IOLTS, containing only the events related to it.

After this *Model Learning* step, we obtain for each component an IOLTS representing its behavior, where each transition is labeled by only one label representing

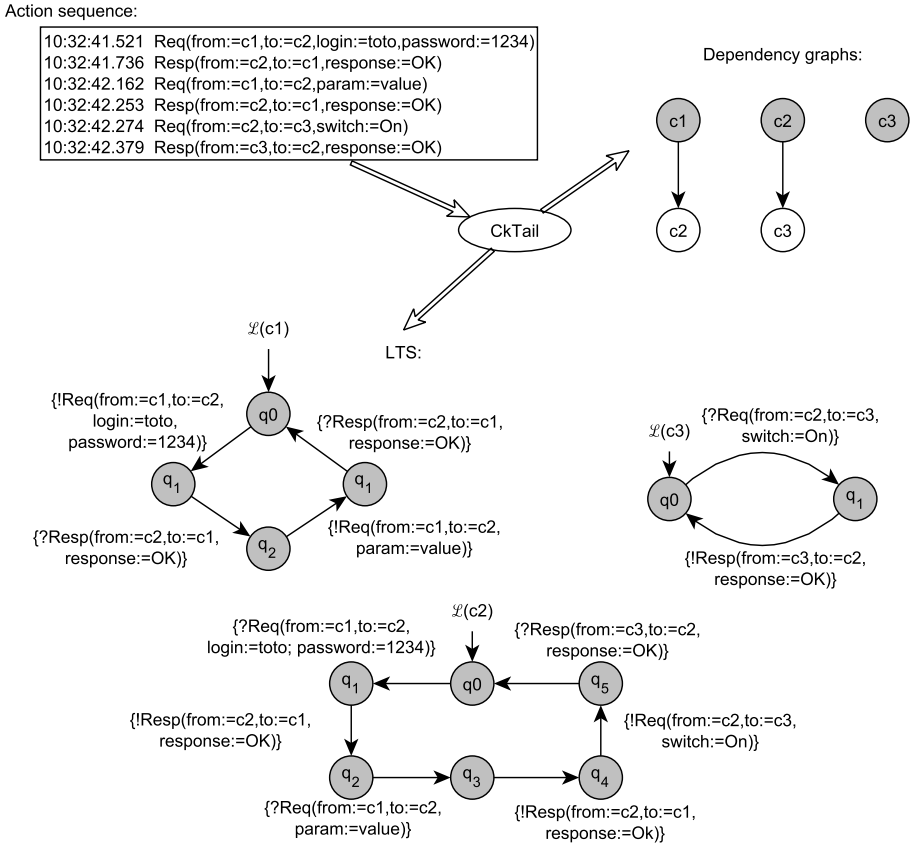


Figure 5.2: Example of result obtained from CkTail.

the action done in the system. However, before verifying any properties on these models, we need to improve them in order to make them appear in the transitions, the propositions we used in the property types. That is the goal of the next step of SMProVer, which injects in the models these propositions.

5.4.2 Model Completion

The goal of the method is to verify some security properties already defined by the user on the model of every component of the system. This can be done only if the models and the properties defined share the same propositions and predicates. The previous step has produced the models of the components, but these models are incomplete as their labels do not contain the propositions and predicates used in the property types, meaning that the property types cannot be verified on the models as they are. That is why the goal of the second step of SMProVer, called *Model Completion*, aims to complete the models by injecting these propositions and instantiated predicates in the labels of the models. However, this injection

can be long and difficult, as it necessitates analyzing the transitions of the IOLTSs to determine if each proposition or predicate can be put on them. To ease this process, we propose to use an expert system that applies transformation rules to the transitions. The rules have to be written by an expert but can be reused on several systems once they are written. The knowledge base of the expert system is the set of transitions of the model $\mathscr{L}(c_i)$, denoted $\rightarrow_{\mathscr{L}(c_i)}$. The transformation rules are represented with the following format: *when* conditions hold on the transitions, *then* actions on the transitions.

```
rule "authentication"
  when
    t1: Transition(isRequest(), contains("login"), contains("password"))
    t2: Transition(isResponse(), isValid(),
        sourceState(t2)=targetState(t1), from(t2)=to(t1), from(t1)=to(t2))
  then
    t1.addKeyWord("loginAttempt(" + getContact() + ")");
    t2.addKeyWord("authenticated(" + getContact() + ")");
end

rule "validResponse"
  when
    $t: Transition(isResponse(), isValid())
  then
    $t.addKeyWord("validResponse");
end
```

Figure 5.3: Transformation rules example.

Figure 5.3 shows an example of transformation rules we have written and that can be used in the models of Figure 5.2. The first rule called *authentication* is applied when the IOLTS contains two consecutive transitions $t1$ and $t2$, where the first one $t1$ is a request that contains the parameters "login" and "password", and the second one is a response of success of the previous request (status code between 200 and 299 in HTTP). When this rule is applied, the proposition $loginAttempt(c)$ is added to the first transition $t1$ and the proposition $authenticated(c)$ to the second one $t2$, with c the component that communicates with the one modeled (obtained with a procedure $getContact()$ here). This can only be applied if the parameters "login" and "password" appear in the actions of the log during an authentication. The second rule called *validResponse* is applied when the model contains a transition t that represents a response of success (status code between 200 and 299 in HTTP). When this rule is applied, the proposition $validResponse$ is added to the transition t.

Figure 5.4 shows an example of the model $\mathscr{L}'(c2)$ from Figure 5.2, after the completion of the transitions from the expert system. We can see that new propositions are available in the labels, the transitions. For example, the expert system has detected in the transition from q_0 to q_1, denoted $q_0 \rightarrow q_1$ that "login:=toto" and "password:=1234" are sensitive data and used as credentials. We can see that the rules defined in Figure 5.3 were applied in this model, as the proposition

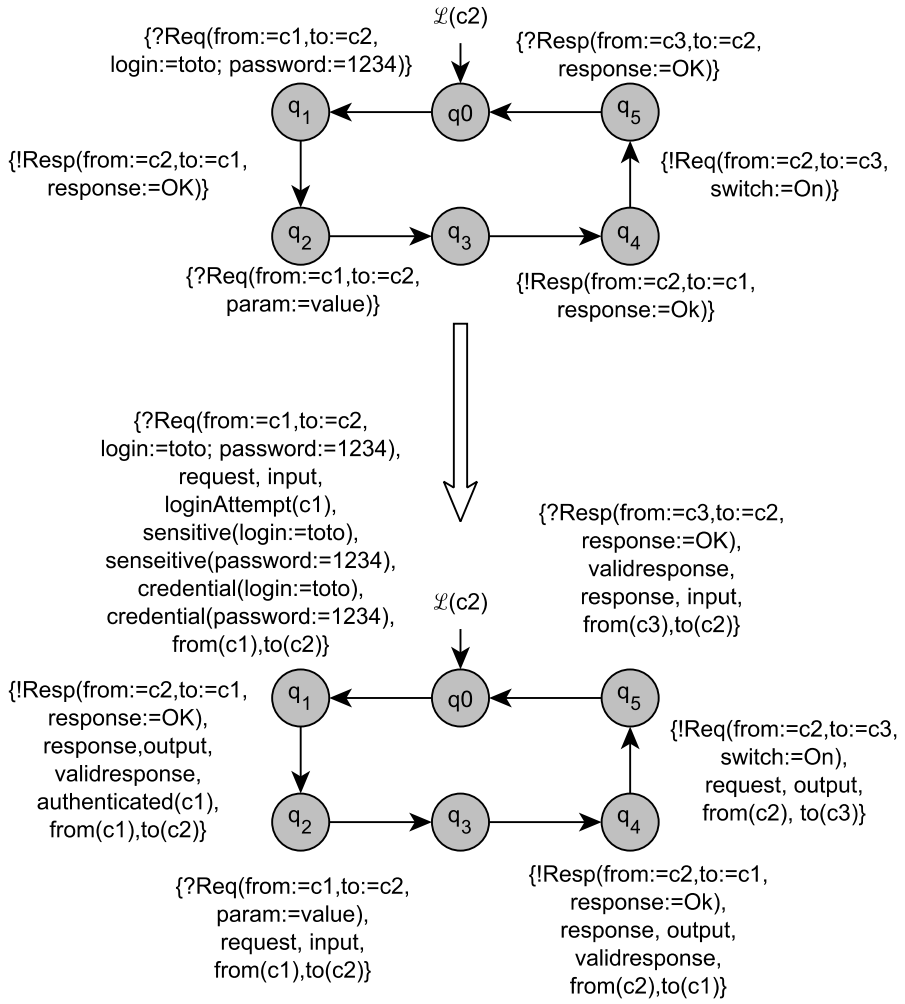


Figure 5.4: Example of LTS completion. New propositions (begin, end, credential, sensitive, validResponse, etc.) are injected on transitions.

$loginAttempt(c)$ appears in transition $q_0 \rightarrow q_1$, $authenticated(c)$ in transition $q_1 \rightarrow q_2$, and $validResponse$ in transition $q_1 \rightarrow q_2$.

At the end of this step, we obtained for each component c_i a new IOLTS denoted $\mathscr{L}'(c_i)$ with new labels in the transitions. However, we advise the user to assess the correctness of the produced models. It can appear that some propositions are missing in the models due to incomplete rules or misinterpretation. For example, an incorrect recognition of encrypted data in the message. It means that some properties cannot be verified on the model. In this case, the method warns the user that some propositions or predicates are missing and that the associated properties cannot be verified or could give wrong results.

The models of components are now ready for verification. However, the predicates of the property types need to be instantiated before to verify them in the model. That is what is done in the next step of SMProVer.

5.4.3 Properties Instantiation

The property types given as input to the method are composed of predicates that should be instantiated before trying to verify them in the models. That is what is done during this third step of SMProVer called *Properties Instantiation*. For each model of component $\mathscr{L}'(c_i)$, this step aims to instantiate the set of property types $\mathscr{P}$ to obtain a set of instantiated properties $\mathscr{P}(\mathscr{L}'(c_i))$ that can be verified on the model $\mathscr{L}'(c_i)$. An instantiated property ϕ is called a property instance and is defined as follows:

Definition 18 (Property instance) *A property instance ϕ of the property type Φ is an LTL formula resulting from the instantiation of the predicates of Φ.*

The property types are instantiated with the help of property bindings, which are functions that associate each variable of the predicates to a possible value in the model.

Definition 19 (Property binding) *Let X' be a finite set of variables $\{x_1, \ldots, x_k\} \subseteq X$. A binding is a function $b : X' \rightarrow Dom(X')$, with $Dom(X') = Dom(x_1) \times \cdots \times Dom(x_k)$.*

Algorithm 13: Property Type Instantiation

input : LTS $\mathscr{L}'(c1)$, property type set $\mathscr{P}$
output: Property instance set $\mathscr{P}(\mathscr{L}'(c1))$

Compute $Dom(deps)$ from $Dg(c1)$;
$X' := \emptyset$;
foreach $\Phi \in \mathscr{P}$ **do**
  foreach $l \in L$ *with* $s_1 \xrightarrow{L} s_2 \in \rightarrow_{\mathscr{L}'(c1)}$ **do**
    if $l = P(v_1, \ldots, v_k)$ *and* $P(x_1, \ldots, x_k)$ *is a predicate of* Φ **then**
      $Dom(x_i) = Dom(x_i) \cup \{v_i\} (1 \leq i \leq k)$;
      $X' := X' \cup \{x_1, \ldots, x_k\}$;
  if X' *is not equal to the set of predicate variables of* Φ **then**
    return a warning;
  else
    $D := Dom(x_1) \times \cdots \times Dom(x_n)$ with $X' = \{x_1, \ldots, x_n\}$;
    foreach *binding* $b \in D^{X'}$ **do**
      $\phi := b(\Phi)$;
      $\mathscr{P}(L'(c1)) = \mathscr{P}(L'(c1)) \cup \{\phi\}$;

This step is implemented by Algorithm 13, which takes as inputs a LTS $\mathscr{L}'(c_i)$ and a set of property types $\mathscr{P}$ and returns a set of property instances $\mathscr{P}(\mathscr{L}'(c_i))$ that can be verified on $\mathscr{L}'(c_i)$. First, the algorithm builds the special domain of the variable $deps$, used with some property types, encoding the notion of dependency among components (line 1). This domain, denoted $Dom(deps)$, gathers the components dependent of $c1$ and is constructed by covering the dependency graph $Dg(c_i)$ of c_i. Then, every property type Φ is instantiated one by one.Algorithm 13 covers the labels of the transitions $q_1 \xrightarrow{L} q_2$ of $\mathscr{L}'(c_i)$ (lines 4-7). If a label $P(v_1, \ldots, v_k)$ is an instantiation of a predicate $P(x_1, \ldots, x_k)$ used in Φ, then the values assigned to the variables $x_1, \ldots, x_k$ are added to $Dom(x_1), \ldots, Dom(x_k)$. The variables $x_1, \ldots x_n$ assigned to at least one value are then added to a variable set X'. Once all the labels of the transitions are covered, the algorithm checks whether all the predicate variables of Φ can be assigned to values (line 8). If it is not the case, the user is warned that the property type Φ cannot be instantiated and has to manually assess it if the security measure associated is implemented by the component c_i (line 8-9). If this is the case, Algorithm 13 computes all the possible bindings of $D^{X'}$ with D the Cartesian product $D = Dom(x_1) \times \cdots \times Dom(x_n)$. Finally, Φ is instantiated by each binding of $D^{X'}$, producing a property instance ϕ which is added to $\mathscr{P}(\mathscr{L}'(c_i))$ (line 12-14).

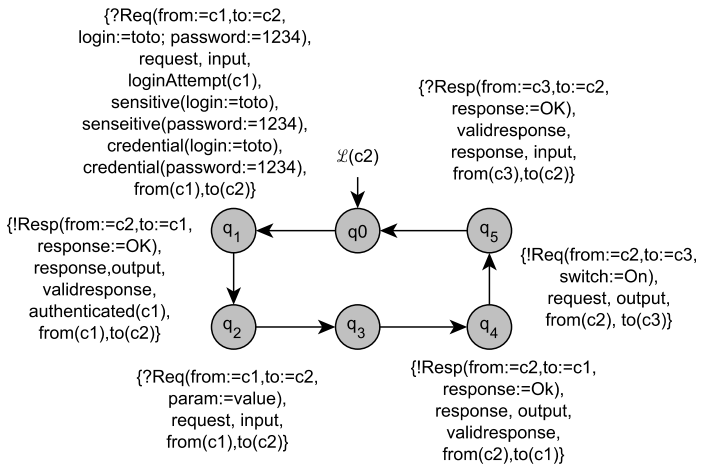
Let illustrate this *Properties Instantiation* step with the property GP-TM-38: $\mathcal{G}(sensitive(x) \rightarrow encrypted(x))$ and the model $\mathcal{L}'(c2)$ of Figure 5.4. On this model, the domain of the variable x is $Dom(x) = \{login := toto, password := 1234\}$. By applying the corresponding bindings, i.e., $\{x \rightarrow login := toto, x \rightarrow password := 1234\}$ on the property type GP-TM-38, we obtain the two following property instances:

- GP-TM-38($login := toto$):
  $\mathcal{G}(sensitive(login := toto) \rightarrow encrypted(login := toto))$
- GP-TM-38($password := 1234$):
  $\mathcal{G}(sensitive(password := 1234) \rightarrow encrypted(password := 1234))$.

5.4.4 Properties Verification

The fourth and last step of SMProVer aims to verify the properties on the models. For each model of component $\mathscr{L}'(c_i)$, the method calls a model checker to check every property instance of $\mathscr{P}(\mathscr{L}'(c_i))$. If all the properties are validated for every model by the model checker, the components implement the security measures, and the system is safe. However, if a counter example is found for a property instance on a model $\mathscr{L}'(c_i)$, that means that the component c_i does not implement the associated security measure, and the system may be vulnerable. The counter example found is then returned to the user, as such a counter example greatly helps the user to investigate where the potential vulnerability in the component c_i is.

Figure 5.5 shows an example of results given by a model checker on the model $\mathscr{L}'(c2)$ of Figure 5.4, with its associated set of property instances $\mathscr{P}(\mathscr{L}'(c2))$. We




```
...

Property GP-TM-24(c1,login:=toto) is false:
counterexample:
-q0 -> q1
  !Req(from:=c1,to:=c2,login:=toto; password:=1234),
  Request,
  output,
  loginAttempt(c2),
  begin,
  sensitive(login:=toto),
  sensitive(password:=1234),
  credential(login:=toto),
  credential(password:=1234)
  from(c1),
  to(c2)
-q1 -> q2
...

Property GP-TM-42(c3) is false:
counterexample:
-q0 -> q1
...
-q4 -> q5
  !Req(from:=c2,to:=c3,switch:=On),
  request,
  output,
  from(c2),
  to(c3)
...
```

```
...

Property GP-TM-38(login:=toto) is false:
counterexample:
-q0 -> q1
  !Req(from:=c1,to:=c2,login:=toto; password:=1234),
  Request,
  output,
  loginAttempt(c2),
  begin,
  sensitive(login:=toto),
  sensitive(password:=1234),
  credential(login:=toto),
  credential(password:=1234)
  from(c1),
  to(c2)
-q1 -> q2
...


Property GP-TM-42(c1) is true

Property GP-TM-53 is true

...
```

Figure 5.5: Example of results. The LTS does not satisfy a property instance related to GP-TM-24 because the login is not encrypted.

can see in this example that the property instance GP-TM-24$(c1, login := toto)$ is invalidated by the model checker, as the first transition $q_0 \rightarrow q_1$ is an authentication attempt with $login; = toto$ as credential. However, this credential is not encrypted, as the predicate $encrypted(login := toto)$ is missing. Similarly, in the first transition the sensitive data $login := toto$ is not encrypted, making the

model checker invalidating the property instance GP-TM-38($login := toto$). We can also see that the property GP-TM-42 is validated by the model checker when it is instantiated with $c1$, as the first two transitions of the model represent an authentication with $c1$. However, this property is not validated when it is instantiated with $c3$, as we can see a communication with $c3$ in the transition $q_4 \rightarrow q_5$, but $c3$ was never authenticated.

5.5 Evaluation

Our approach was implemented in a toolchain that was used to perform some experiments to evaluate our method. The first step, *Model Learning*, is performed by our implementation of CkTail [1] already presented in the previous chapter. The second and third steps of the method, *Model Completion* and *Properties Instantiation* are implemented in a new tool called SMVmaker[2], that takes as inputs a model of a component, a set of property types, and transformation rules, and produces a file representing the model and the property instances. This file can be taken as input by the model checker NuSMV[3], which then performs the last step *Properties Verification* by verifying the properties on the model. With this toolchain, we try to answer the following research questions:

- RQ1 (Sensitivity): can SMProVer detect the security measures implemented by the system?
- RQ2 (Specificity): can SMProVer detect the security measures that are not implemented by the system?
- RQ3 (Scalability): how the execution times scale according to the size of the log and the set of property types given as input?

5.5.1 Setup

In order to perform these experiments, we constructed several use cases of IoT systems described below:

- **UC1** is composed of 3 motion sensors that turn on a light bulb when they detect movements and communicate through a gateway. All of them have user interfaces, which may be used for configuration. The main purpose of this use case is to focus on security measures related to classical attacks (code injection, brute force, availability).
- **UC2** includes a motion sensor, a switch, and a camera. The motion sensor communicates with the switch in clear text and with the camera with

[1] https://github.com/Elblot/CkTailv2
[2] https://github.com/Elblot/SMVmaker
[3] http://nusmv.fbk.eu/

encrypted messages. The camera also communicates with external clouds, requiring authentication and encryption. In this use case, we mainly focus on vulnerabilities and security measures related to encryption and authentication.

- **UC3** is composed of 3 cameras, which communicate with external cloud servers. Two cameras can check for updates and install them. This use-case aims at focusing on vulnerabilities and security measures related to the update mechanisms.

- **UC4** is composed of a thermometer that communicates with a gateway. A user communicates with the gateway and tries to perform some attacks (XSS). In this use case, we mainly focus on vulnerabilities and security measures related to encryption, authentication, and code injection.

- **UC5** is composed of a camera that communicates with an NTP server, an SMTP server, and an FTP server. The camera needs to authenticate with encrypted credentials with the FTP server and with credentials in clear with the SMTP server. A user can communicate with the camera over HTTP. In this use-case, we mainly focus on vulnerabilities and security measures related to encryption and authentication.

For each use case, a first model is generated from a log collected in a normal condition (without attack attempts). The model of each component is then analyzed in order to find which one is testable. Then a second log is collected from the systems, where penetration testing tools are used on the components that are testable. This was possible only for use cases $UC1$, $UC4$ and $UC5$ as we do not have access to the systems of $UC2$ and $UC3$, where only the logs were available. The models produced for each configuration are large (from 168 to 1416 states); analyzing them manually or producing manually security properties that can be verified on them is a long and difficult task.

The experiments were performed on a desktop computer with 1 Intel(R) CPU i5-7500 @ 3.4GHz and 32GB RAM. The transformation rules used by our expert system are given in the appendix.

5.5.2 RQ1 Sensitivity

Procedure: In order to answer this research question, we used our tools chain to verify on each use case that the security measures represented by the property types described in Section 5.2 are implemented by every component of the use cases. The results obtained from our tools chain were then manually analyzed to determine the rate of property instances validated by the solver, that were correctly evaluated (Sensitivity).

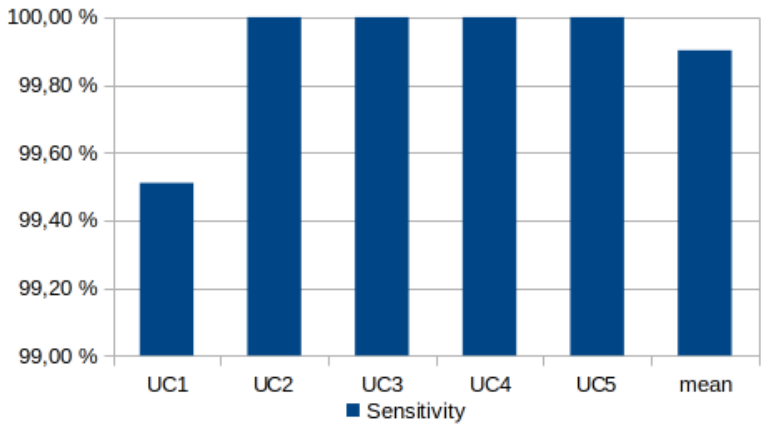


Figure 5.6: Sensitivity measured for each use case.

Results: Figure 5.6 shows the sensitivity measured for each use case. We can see that the rate of property instances validated by the solver correctly evaluated is 100% for $UC2$ to $UC5$. However, it is not the case for $UC1$, where we measured a sensitivity of 99.51%. We observed that some rules of the expert system were not precise enough and could give false results. For example, the rule that determines if data is encrypted calls a tool that calculates the message entropy. Unfortunately, this method is not precise on small data, but we did not find any better solution in the literature. We advise the user to manually verify that the expert system completes the models correctly and to correct the models if that is not the case. The other incorrect evaluations are due to the over-generalization or under-generalization of the models produced by CkTail. For example, the property GP-TM-25 described in Section 5.2, expressing that an account is locked after 5 failed authentication attempts in a row, needs that the 5 attempts are processed during the same session. In $UC1$, the attempts are separated in different sessions, making that the tool validates the property when it should be invalidated in reality. All the passive model learning methods lead to an imprecise model of the system, including CkTail. However, we have shown in the previous chapter that CkTail produced more precise models than other passive model learning methods designed for communicating systems. These results tend to show that the method succeeds in finding that security measures are implemented in an IoT system.

Moreover, we have observed during this experimentation that a lot of property instances were produced. in order to correctly verify that the system implements all the security measures. Between 37 and 823 property instances were produced for each use case, and the model generation and the property instance generation never took more than half an hour in our use cases. We believe that it would take a lot more time to manually produce the models or to manually produce the LTL properties to verify on an existing model. It tends to show that generating the model and adapting automatically the property to verify on them greatly ease the analysis

of the security measures implemented by the system.

5.5.3 RQ2: Specificity

Procedure: In order to answer this research question, we used our tools chain to verify on each use case that the security measures represented by the property types in Section 5.2 are implemented by every component of the use cases. The results obtained from our tools chain were then manually analyzed to determine the rate of property instances invalidated that were correctly evaluated (Specificity).

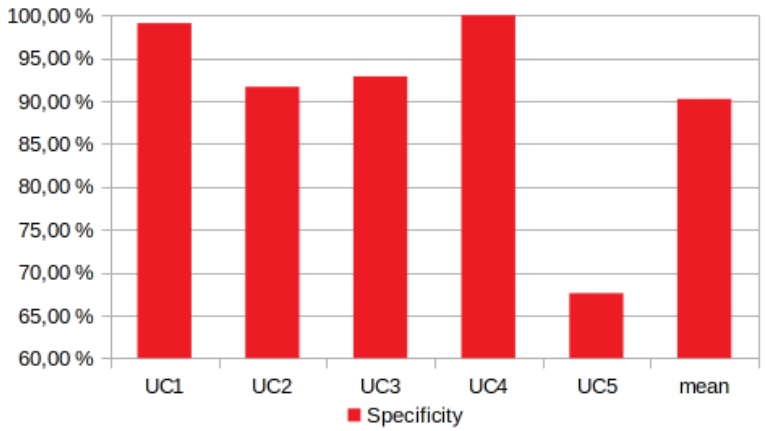


Figure 5.7: Specificity measured for each use case.

Results: Figure 5.7 shows the specificity measured for each use case. The overall rate of property instances invalidated that are correctly evaluated is 90.23%. Once again, we observed that some rules of the expert system could give false results due to their imprecision. It is especially the case for the rule that determines if a data is encrypted; that gives a lot of false results for $UC5$, explaining the lower specificity for this use case. We also find in one of our use cases that the authentication phases are encrypted, making it difficult to detect them, even manually by the user. If the labels are verified manually, the specificity of $UC5$ increases to 89.19%. However, even if the user verifies the labels on the transitions of the model, the specificity does not reach 100% for most of our use cases. We observed once again that the over-generalization or under-generalization of CkTail may cause false results during the verification of the properties. For example, some behaviors can be separated from previous authentication in the model, making the system allow to do them without authentication in the model, invalidating the associated property. The two first research questions show that SMProVer is precise, as it can detect the security measures that are implemented in a system, and it can detect issues more precisely when the labels of the transitions are correctly completed.

5.5.4 RQ3: Scalability

Procedure: In order to answer this research question, we used our tools chain to assess the security of an IoT system, varying the size of the inputs given to the method. In a first time, we made the size of the log given as input varying, from 500 events to 10000 events, using the same number of property types (the 11 described in Section 5.2), and we measured the time taken by our tools chain to return the verdict in order to see the effect of the size of the log on the efficiency of our tools chain. In a second time, we made the number of property types given as input to our tools chain varying, using the same log, and we measured the time taken by our tools chain to see how the efficiency is affected by the size of the set of property types taken as input.

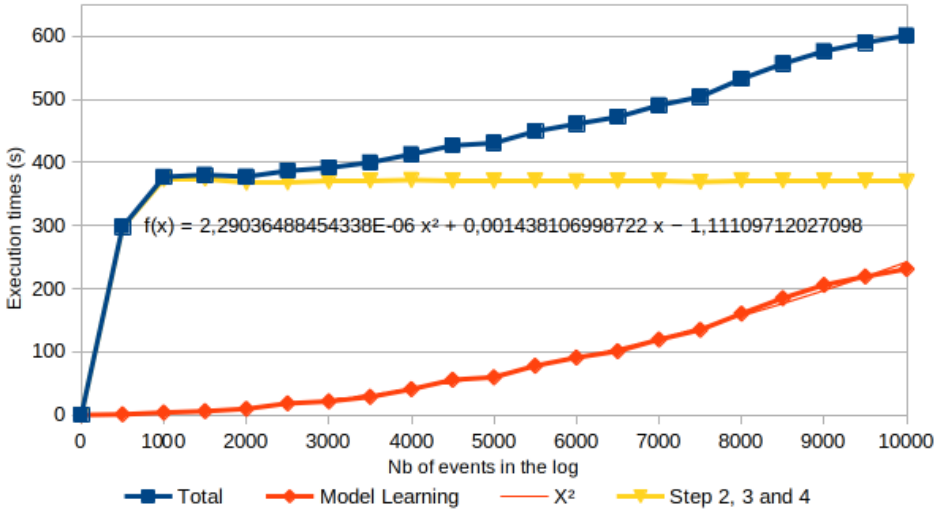


Figure 5.8: Evolution of execution time according to the number of events in the log.

Results: Figure 5.8 shows the evolution of the execution time of our tools chain, according to the number of events in the log given as input to the method. In this figure, the execution time of the first step of the method is separated from the other steps. We can first see that the time complexity of our model learning method, CkTail, is quadratic, as stated in Chapter 4. We can also see in this figure that the execution time of the other steps first sharply increases when we have between 0 and 1000 events in the log and then becomes constant with more than 1000 events in the log. After analyzing the models produced, we have determined that the models produced with more than 1000 events contain the same number of behaviors of the system and have the same size, while the models produced with only 500 events contain fewer behaviors and are smaller. It is not surprising that the models produced using a lot of events have the same size as the majority of the

behaviors of our IoT systems are cyclic and repeat themselves, meaning that for big log, increasing the number of events in the log is less likely to increase the number of different behaviors in the models and then the size of the models. The evolution of the execution time of the different steps of the method explains the evolution of the execution time of the complete method. First, the execution time increases greatly as the size of the models produced by CkTail is increased too. Then, once the models do not grow anymore, only the execution time of CkTail is affected by the size of the log, and the method takes a quadratic complexity. CkTail is able to take large logs as input, and we believe that SMProVer can then also take a large log of a system to assess its security.

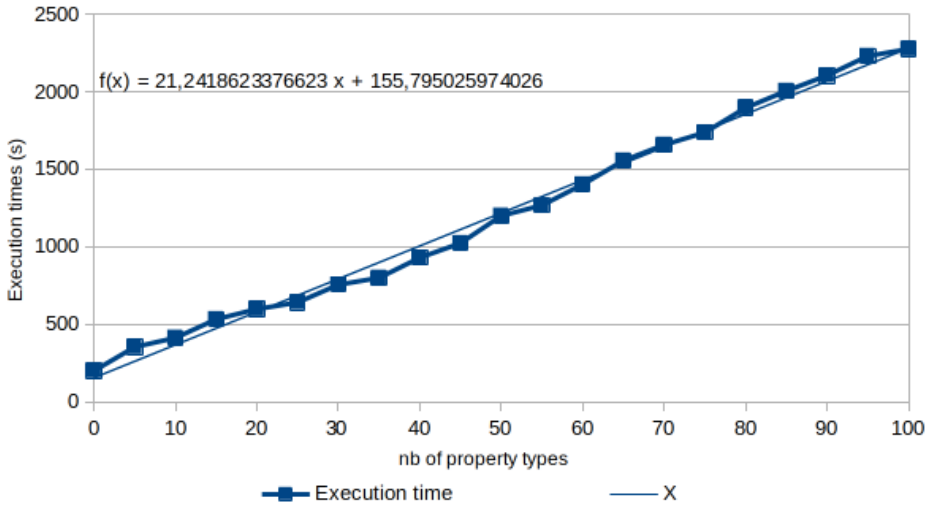


Figure 5.9: Evolution of execution time according to the number of property types given as input.

Figure 5.9 shows the evolution of execution time according to the number of property types given as input to the method. The properties given as input take all the same time to be processed and verified in order to avoid any bias. Even if no property type is given to our tools chain, the model is still generated by CkTail, transformed by the expert system, and loaded by the model checker, taking approximately 200 seconds in our use case. In the other measures, it also takes approximately 200 seconds to process the model, meaning that the rest of the time is taken by our method to instantiate the property types and verify the property instances in the models. We can see that the time complexity of the method is linear according to the size of the set of property types given as input, meaning that the method can take a lot of property types to verify them on the system. In our cases, the method could even take 100 property types and takes less than 40 minutes to verify all of the instances produced from them on every component of the system.

5.5.5 Threats to Validity

Some threats to validity can be identified in this evaluation. First, only five use cases were considered in this experimentation. It is a threat to validity because the results obtained are different from one use case to another. (the specificity goes from 67% to 100%), making it difficult to draw a general conclusion from the results obtained. A second threat to validity is linked to the rules used by the expert system that may be imprecise, as observed during the experimentation, especially the rule determining if data are encrypted. It is a threat to validity because the imprecision of the rules impacts the sensitivity and specificity of the method, as described in RQ1 and RQ2.

5.6 Limitations

This method suffers from some limitations. First, SMProVer is dependent on CkTail, as CkTail is used by SMProVer, meaning that SMProVer also suffers from the same limitations as CkTail, described in the previous chapter and recalled next. CkTail is a passive model learning method, meaning that the precision of the models produced depends on the size and content of the log given as input. The assumptions made on the log are realistic but do not take into account all the IoT systems. Moreover, a threshold is needed by CkTail in order to produce precise models, which can be difficult to choose.

An other limitation of SMProVer is linked to the difficulty of writing the property types given as inputs to the method and the transformation rules used by the expert system. Even if we greatly reduce the number of LTL properties that had to be written by generalizing them to all the systems, it can still be difficult to write such properties. A first set of property types is given in this thesis; however, this set is only built from one recommendation database and only represents a part of it. We plan in the future to improve this set of property types with more properties, representing more security measures from other sources, like for example the Owasp [45]. The transformation rules used by the expert system had to be written by an expert besides the property types. These rules depend on the system, and it can be difficult to write precise rules. We have observed this during our experiments, where some of our rules were not precise enough to complete efficiently the models, such as in the case of the rule determining if data are encrypted. That is why we advise the user to verify manually the models completed by the expert system, but this verification is difficult.

Finally, an other limitation of the method is that the components are only assessed in isolation. Even if the model of a component contains the interactions of the component with the other components, the method cannot be used to assess complex interactions involving more than two components, like it is the case for several security protocols, as the model of a component does not contain the interactions of the other components, and the models are assessed in isolation. In the future, we plan to investigate if such analysis can be performed using the models

produced.

5.7 Summary

In this chapter, we presented SMProVer, a method that aims to help an auditor verify that every component of a system implements security measures, expressed with the help of generic LTL properties called property types. The method combines model learning and model checking to produce the models of the different components of the system and verify some properties on them. The models are first produced with the help of the CkTail method, described in Chapter 4, and are then transformed by SMProVer in order to complete them with new labels, making the verification of properties on them possible. The properties are then instantiated to make them verifiable on the models produced and are finally verified on them with the help of a model checker.

The results of the experiments tend to show that our method succeeds in finding the majority of the security measures expressed in the LTL formula. They also show that the issues are found in a short time, showing the usefulness of the method to help an auditor detect security issues. Finally, the experiments show that the method can take a large log as input and can take a large set of property types representing the security measures to verify.

We plan to improve our method by making it easier to use. Currently the model completion step is performed by an expert system needing rules written by an expert that has a lot of knowledge on the system. Producing such rules can be a hard task, and some of these rules can be imprecise, making that the user is advised to verify manually that the new labels are correctly put in the models. We plan to investigate if adding some restrictions in the system may ease this completion and allow us to make it fully automated without the needing of a verification by the user. We also plan to produce a database of generic transformation rules that can be applied to several communication protocols used by IoT systems that could be used by the auditor, reducing the knowledge needed in the system.

This work was published in conference papers:

- Blot Elliott and Sébastien Salva. Verification de recommandations sur les objets connectes. In *19èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL 2020,* Vannes, France, 2020.

- Sébastien Salva and Elliott 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*, pages 350–360, 2020.

Chapter 6

Conclusion

6.1 Summary

Some IoT device providers do not check the security of the devices produced, as it can be a long and costly task. Consequently, several large-scale attacks in the past targeted these devices with well-known vulnerabilities. That is why, in this work, we focused on the automation of a part of the security audit of IoT systems.

Four methods were proposed in this thesis to help an auditor assess the security of an IoT system. Three of them aim to produce the models of the components of a system that give a clearer view of the system and that can be assessed with a formal method. The last one aims to assess the security of the components of a system by assessing their models generated automatically and applying formal methods to verify some properties on them.

The contributions were the following:

- COnfECt and ASSESS are two passive model learning methods designed for component-based systems that produce for each component of the system a model representing its behavior. The methods take as input a set of raw events and produce an LTS for each component of the system. Both methods can be used to produce the model of a component-based system where the communications among the components are hidden, like, for example, a complex embedded device composed of several sensors and actuators. The first method, COnfECt, is more general and can be applied to more systems than ASSESS. However, COnfECt is harder to use than ASSESS, as the user needs a lot of knowledge on the system to define different factors. ASSESS is simpler to use but cannot be used on all component-based systems, as the events given as input have to include an identifier of the component that produced them.

- CkTail, a passive model learning method designed for communicating systems. CkTail takes as input a log and produces for each component of the system an IOLTS representing its behavior and a dependency graph repre-

senting how the component interacts with the others. This method builds more precise models than the other methods designed for communicating systems by detecting sessions in log. Two different strategies are proposed to detect sessions. The first one, CkTail-w/S, supposes that the events of the log given as input contain an identifier of the session. The second one, CkTail-w/oS, supposes instead that the components follow strict behavior.

- SMProVer, a method using model learning and model checking to verify that every component of the system implements security measures. The method takes as input a log extracted from the system and a set of generic LTL properties containing variables, called property types, that represent the security measures to verify. The method aims to produce the models of the components with the help of CkTail and to add in their transitions the proposition used by the property types using transformation rules. The property types are then instantiated according to the values that can be found in the models in order to make the properties verifiable on the models. Finally, the properties are verified on the models with the help of a model checker. We also propose a set of property types that can be used to cover a part of security measures found in the literature and a set of transformation rules that can be used on an HTTP-based system.

Each of these methods was evaluated with the help of implementations available as open source[1]. These evaluations show the usefulness of our methods, showing their performance and precision of the model generated or the detection of security flaws. The evaluations showed that our methods could be used in real cases, and we believe that they could be used even with little knowledge of the systems, allowing IoT device providers to use them to improve the security of their products with a reduced cost and time.

6.2 Perspectives

6.2.1 Improvement of the Methods

The methods proposed in this thesis still have room for improvement. COnfECt can be really difficult to use as it requires that the user has a lot of knowledge on the system to model. We tried to make the method ASSESS, which is an adaptation of COnfECt, easier to use by reducing the knowledge needed on the system. However, by doing so, we added a new assumption on the system, making that only systems using component identifiers in logs can be modeled by ASSESS. In the future, it can be interesting to find other ways to determine if several events are produced by the same component. It can also be interesting to find an automated way, by assessing the log, to determine which factors the user has to use with COnfECt for the system. Moreover, several strategies are provided with COnfECt

[1] https://github.com/Elblot

and ASSESS, each aiming to model an architecture of the IoT system. The strategies can model, for example, a system where the components interact with each other or where they run in parallel without interaction. However, few strategies are proposed (3 for COnfECt and 2 for ASSESS), and they may not represent all the possible architectures. In an IoT system, some components can interact with each other, and some others can run in parallel without any interaction. Such configurations cannot be modeled by the current synchronization strategies provided by the methods. In the future, we plan to study the creation of new synchronization strategies that allow to model different architectures of IoT systems.

CkTail relies on several assumptions in order to efficiently detect sessions. Two strategies are proposed for this session detection. The first one relies on the identifier of the session present in the events of the log. However, we observed that this kind of identifier is not always used in IoT systems. That is why a second strategy was designed, relying on the strict behavior of the components. However, even if an identifier is no longer required, CkTail cannot produce the model of every system with this strategy as the behavior of the components is restricted. In the future, we plan to study how to make CkTail usable on more systems by relaxing some assumptions.

The method SMProVer can also be improved. SMProVer uses an expert system requiring transformation rules to add labels in the models of the components. These rules have to be written by an expert that has knowledge of the system. However, the user has to manually verify that the labels are correctly put in the models, making that the method is not fully automated. In the future, we plan to study if we can automate the method totally by adding new requirements to the system under analysis. Moreover, the rules used by the expert system and the property types representing security measures given as input can be difficult to write. In the future, we plan to produce more property types and transformation rules in order to make a database that could be used by non-experts.

6.2.2 New Approach

A new approach is currently in development that helps a user produce mocked IoT devices. A mock is a test-specific device that mimics the behavior of a real device of the system and replaces it. Using such devices allows the user to have more control during the test of the system and is particularly useful when they replace devices that cannot be controlled, like sensors. The method addresses the generation of such mock IoT devices. The models of the components that can be generated with the help of CkTail are assessed in order to determine which components should be replaced by mocked ones.

The contributions will be the following:

- Classification of the components with the help of quality metrics in order to choose if the components can be tested in the system, tested in isolation, replaced by a mock, or only code review is possible on them. The quality

metrics used are the following:

  - $Understandability$: evaluates how much component information can be interpreted and recognized [3, 43].
  - $Accessibility$: evaluate how many access points of the component can be reached.
  - $Dependability$: helps better understand the architecture of a system. This metric can be separated into two different metrics: $InDeps$, which evaluates the degree to which a component is needed by the other, and $OutDep$, which evaluates the degree to which a component requires other components for functioning.
  - $Observability$: evaluates how the specified inputs affect the outputs [20].
  - $Controllability$: refers to the capability of a component to reach one of its internal states by means of a specified input that forces it to give a desired output.

- We propose an algorithm that helps to generate the mocked device, in the form of a mock model, from the behavioral model of the devices. We will also propose an algorithm called mock-runner that takes as input a mock model and executes it.

- An implementation of the quality metrics analysis of the models of the components, and an implementation of the mock-runner taking as input a DOT file containing the mock model representing the behavior of the component.

With the help of the metrics, we classify each component into at least of the following categories: "Mock", "Test", "Test in isolation" and "Code review". The components in the "Mock" category are the components that can be replaced by a mock in order to improve the test of the system. The components in this category are accessible (high $Accessibility$) and needed components (high $OutDeps$), or accessible (high $Accessibility$), uncontrollable (low $Controllability$) and dependent-only (high $InDeps$) components. In the first case, the mock will be used in order to test how dependent components interact with it. In the second case, the component is replaced by a mock because it cannot be tested in the system.

The components in the category "Test" or "Test in isolation" are the components that can be tested, respectively, in the system by interacting with the other components or in isolation. The components in these categories are the components that are testable, i.e., components that are controllable (high $Controllability$) and observable (high $Observability$). Moreover, the components in the category "Test in isolation" are the components that do not need the other components for functioning (minimal $OutDeps$).

Finally, the components in the "Code review" category are the ones that cannot be tested because they are not accessible (low $Accessibility$), not understandable (low $Understandability$), or testable (low $Controllability$ or $Observability$).

Currently, a first preliminary version of the analysis of the models of the component was implemented as an open source [2] that takes as input a model of a component and measures its $Observability$ and $Controllability$. It is planned to improve this implementation to measure the other quality metrics. Moreover, a first version of the mock runner was implemented and is available on github[3]. This implementation still needs improvement; in the future, we plan to improve it and use it to evaluate our approach. Moreover, we also plan to improve the mock generation by applying several mutations to the messages sent by mocked devices in order to reproduce attacks and better test the security in the system.

6.3 List of Publications

- Elliott Blot, Patrice Laurencot and Sébastien Salva. COnfECt : Une Méthode Pour Inférer Les Modèles De Composants D'un Système. In *17èmes journées AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, Grenoble, France, June 2018.

- Sébastien Salva and Elliott 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.*, pages 298–305, 2018.

- Sébastien Salva, Elliott Blot and Patrice Laurençot. 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*, pages 142–148, 2018.

- Sébastien Salva and Elliott Blot. Reverse engineering behavioural models of iot devices. In *31st International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Lisbon, Portugal, July 2019.

- Sébastien Salva and Elliott Blot. Model generation of component-based systems. *Software Quality Journal*, 28(2):789–819, January 2020.

- Sébastien Salva and Elliott 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, pages 27–38.

[2] https://github.com/Elblot/testability
[3] https://github.com/Elblot/Mock-server

- Blot Elliott and Sébastien Salva. Verification de recommandations sur les objets connectes. In *19èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL 2020,* Vannes, France, 2020.
- Sébastien Salva and Elliott 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*, pages 350–360, 2020.

Chapter 7

Appendix

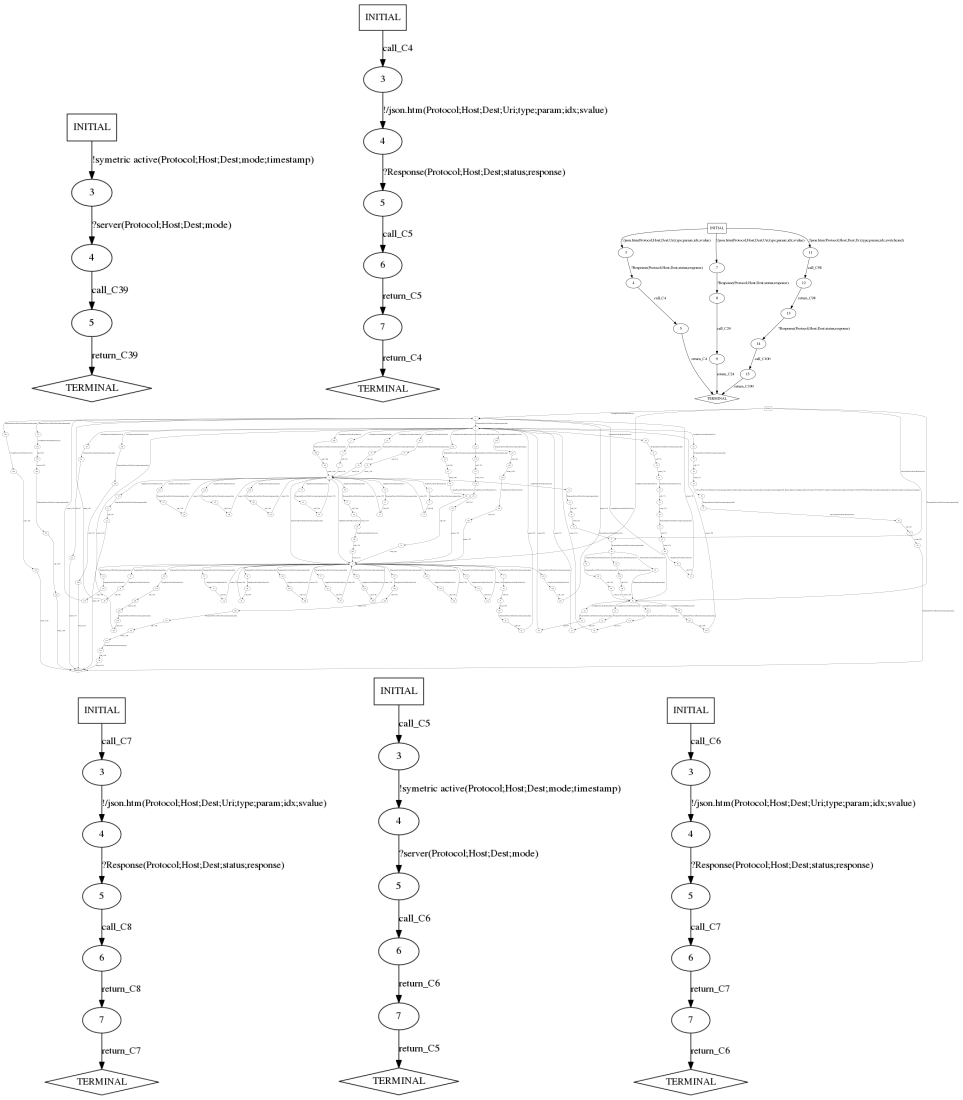


Figure 7.1: Example of some models obtained using COnfECt with the Strict strategy. Only a portion of models are shown as more than 100 models are produced.

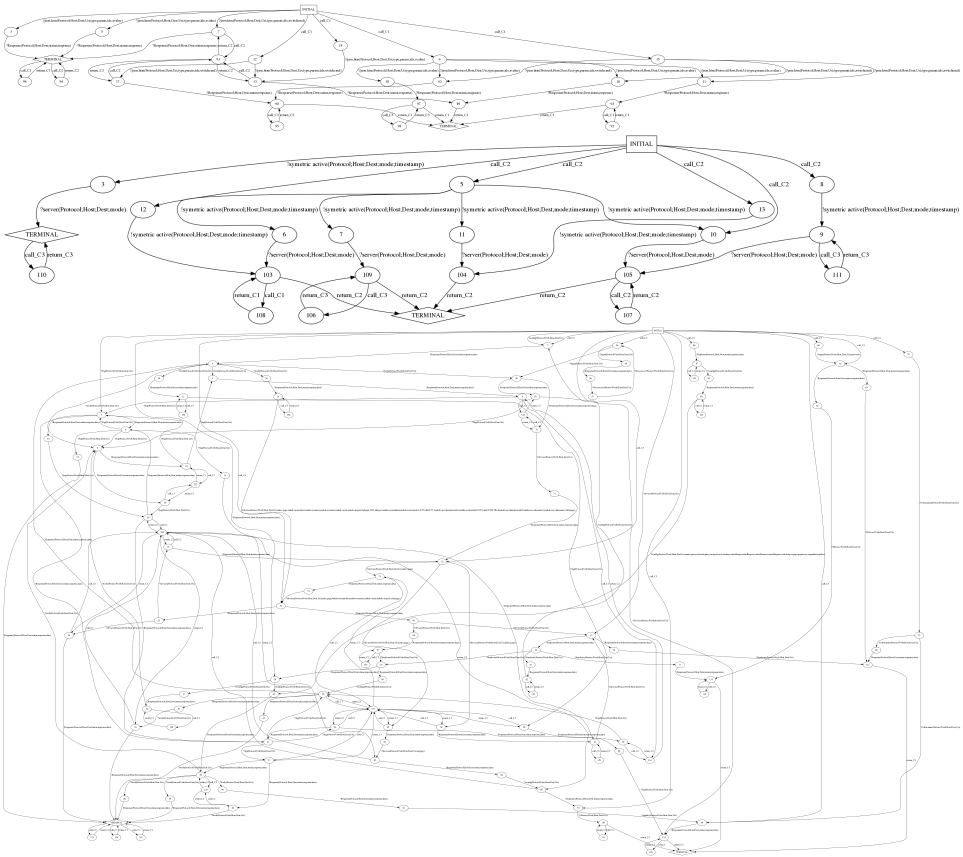


Figure 7.2: Example of some models obtained using COnfECt with the Weak strategy.

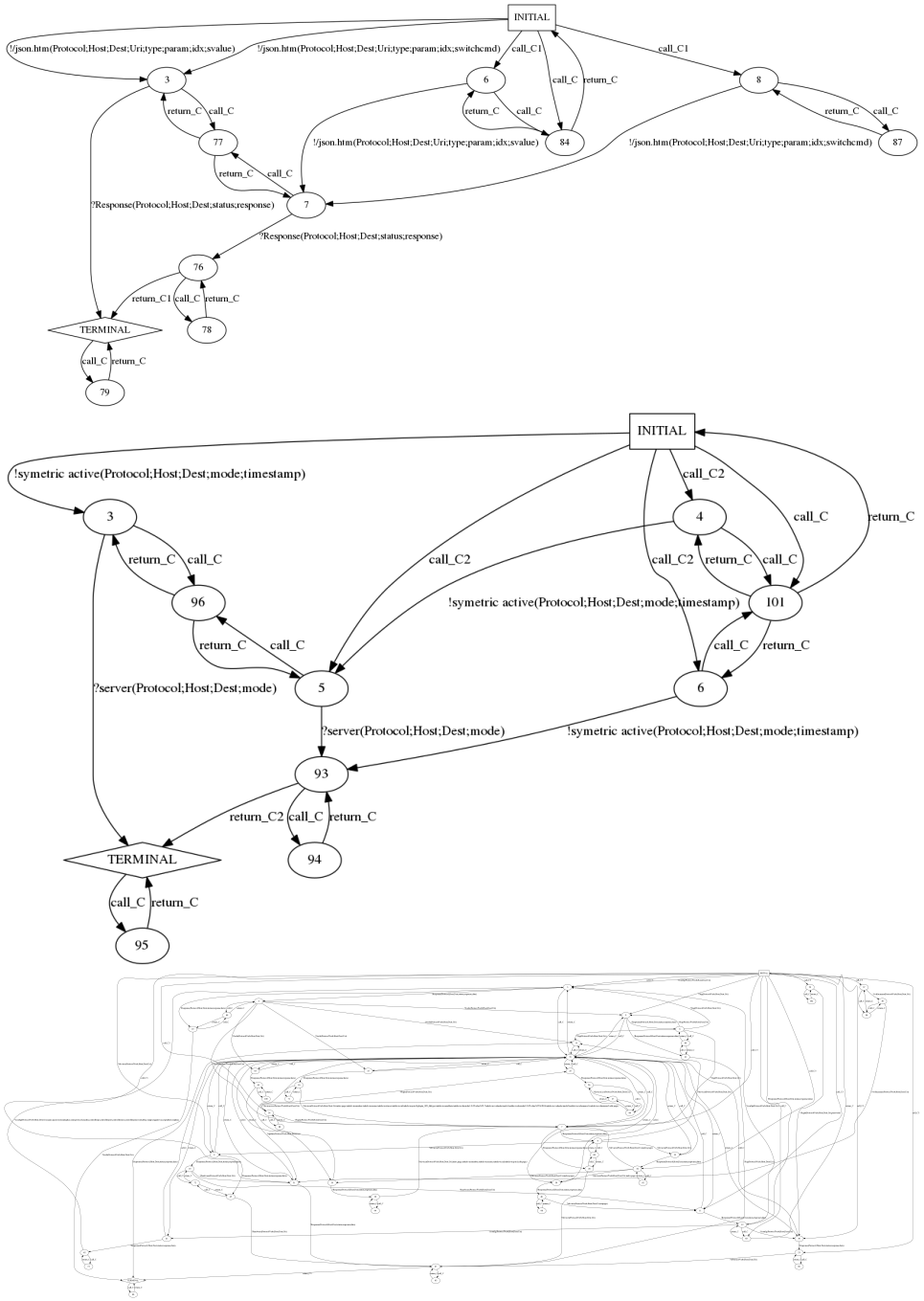


Figure 7.3: Example of some models obtained using COnfECt with the Strong strategy.

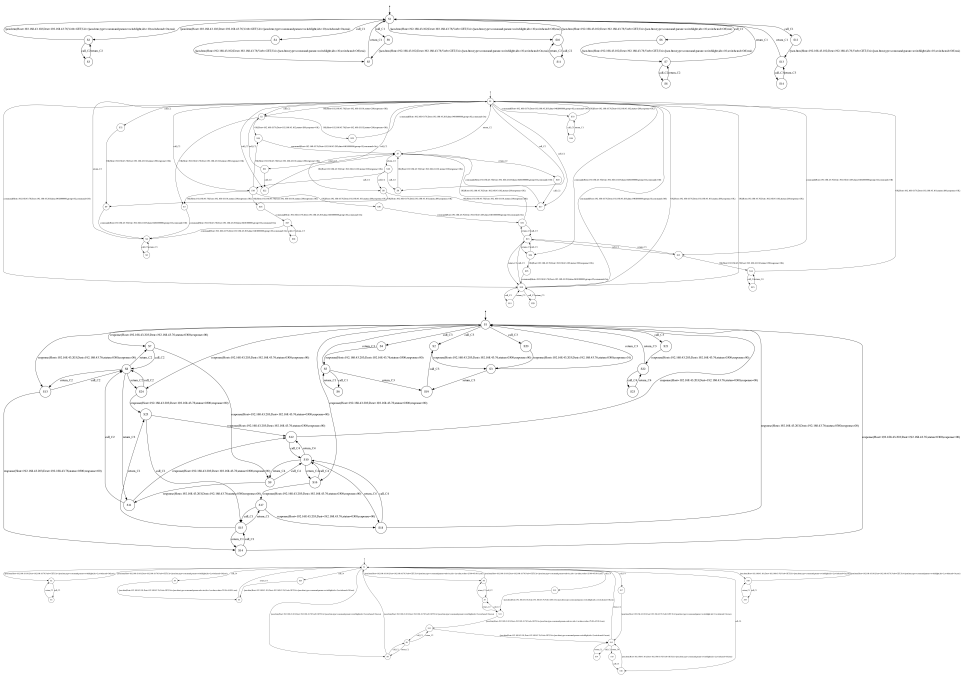


Figure 7.4: Example of some models obtained using ASSESS with the Loose-coupling strategy.

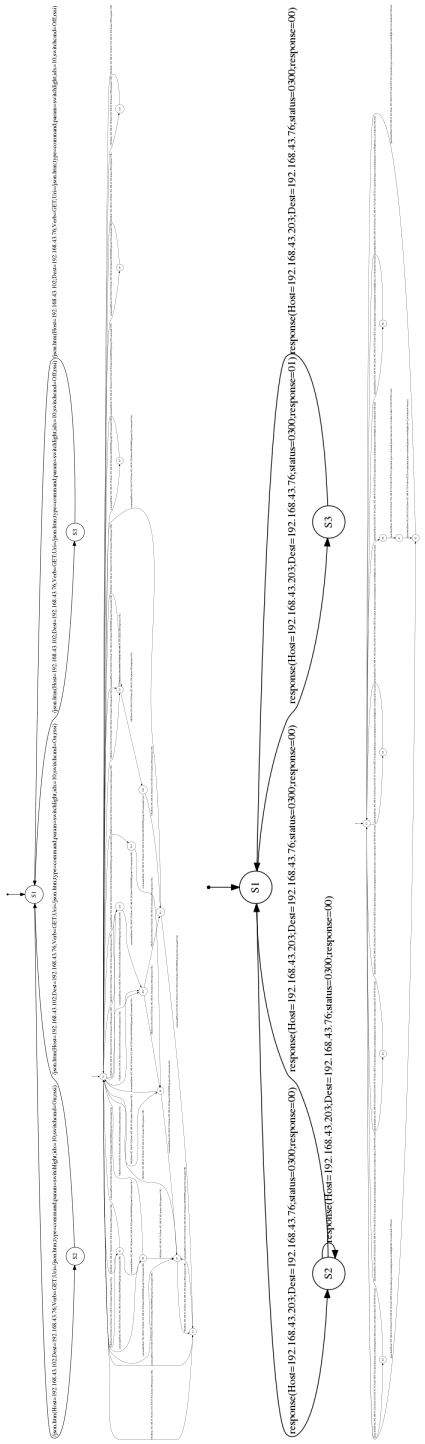


Figure 7.5: Example of some models obtained using ASSESS with the Decoupling strategy.

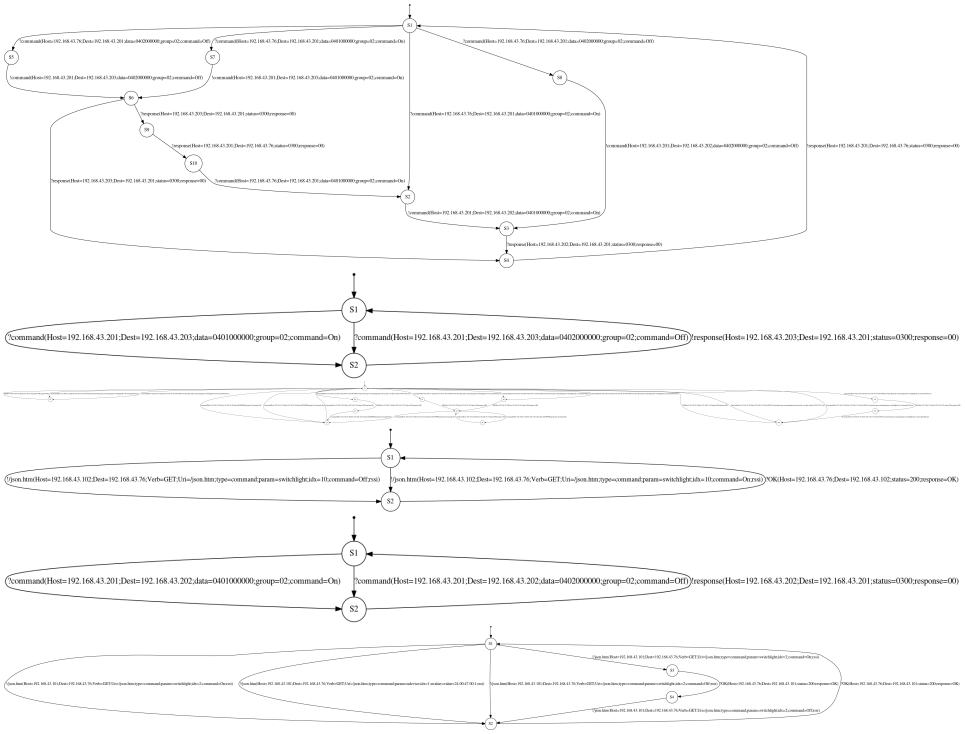


Figure 7.6: Example of some models obtained using CkTail.

```
1  begin|An initial state, denote the potential beginning of a new
       session|this label cannot be missing in the system
2  end|An initial state, denote the potential end, of a session|this label
       cannot be missing in the system
3  from_c|denote the source of the message |this label cannot be missing in
       the system
4  to_c|denote the destination of the message|this label cannot be missing
       in the system
5  request|the message is a request|this label cannot be missing in the
       system
6  response|the message is a response|this label cannot be missing in the
       system
7  input|the message is an input|this label cannot be missing in the system
8  output|the message is an output|this label cannot be missing in the system
9  getUpdate_d|denote the sending of a new update file|(1)The components
       cannot receive update (2)no update occurred in the log -> check if
       device can receive updates
10 cmdSearchUpdate|the order is given to search for a new update|this label
       cannot be missing in the system
11 sensitive_d|sensitive data are present in the message|this label cannot
       be missing in the system
12 credential_d|credentials are present in the message|this label cannot be
       missing in the system
13 encrypted|denote the data are encrypted|this label cannot be missing in
       the system
14 searchUpdate_d|denote the component search for a new update|(1)the
       component cannot search a new update (2)the component doesn't search
       for update in the log -> check if device can search for updates
15 loginAttempt_c|denote an authentication attempt|(1)there is no
       authentication to exchange data with the component (2)the
       authentication phases are encrypted (3)there is no authentication in
       the log -> check if authentication needed to communicate with the
       component
16 authenticated_c|denote the success of an authentication phase|this label
       cannot be missing in the system
17 loginFail_c|denote the failure of an authentication phase|(1)there are no
       failure of authentication in the log -> check if authentication can
       fail
18 lockout_c|the account id locked|(1) account cannot be locked (2) account
       are not locked in the log -> check if authentication is weak against
       brute force attack
19 passwordRecovery|user try to recover a forgotten password|(1)user cannot
       recover forgotten password (2) user doesn't try to recover password
       in the log -> check if password recovery system(if it exist) doesn't
       give to much information about existing users
20 blackListedWord|denote a black listed word in error message|this label
       cannot be missing in the system
21 validResponse|the response is OK|no valid response
22 errorResponse|the response is not OK|no error response
23 unavailable|the component is unavailable|never unavailable in the model
24 XSS_d|the request contain code injection|(1) the requests don't contain
       code injection in the log -> check if the web interface (if exists)
       is resistant to code injection
25 SQLinjection_d|the message contains an SQL injection|(1)the system
       contains code injection that doesn't match the ones implemented (2)no
       SQl injection attempt was made during the log collection
```

Figure 7.7: Keywords used by SMProVer.

```
1  GP-TM-18;;;G((getUpdate_*getUpdate*) -> (encrypted &
       !(sensitive_*getUpdate*))) & G ((begin & F end) ->
       (!((getUpdate_*getUpdate*) & (from_*compo*)) U
       ((authenticated_*compo*) | end)));;;The device can update, and the
       update file is encrypted, doesn't contain sensitive data, and the
       source is authenticated.
2  GP-TM-19;;;F searchUpdate_*searchUpdate* -> !(F
       searchUpdate_*searchUpdate* -> (!searchUpdate_*searchUpdate* U (input
       & cmdSearchUpdate & ! searchUpdate_*searchUpdate*)));;;The device can
       automatically search for updates
3  GP-TM-24/GP-TM-40;;; G(((loginAttempt_*compo*) &
       (credential_*credential*)) -> encrypted) ;;;data are encrypted during
       authentication phase,credential are not exposed in the traffic
4  GP-TM-25;;;G ((begin & F end) -> (!( G ((begin & F (end |
       (authenticated_*compo*)))-> (( ! (loginFail_*compo*) & !(end |
       (authenticated_*compo*))) U (end | (authenticated_*compo*) |
       (((loginFail_*compo*) & ! (end | (authenticated_*compo*))) U (end |
       (authenticated_*compo*) | (( ! (loginFail_*compo*) & ! (end |
       (authenticated_*compo*))) U (end | (authenticated_*compo*) |
       (((loginFail_*compo*) & ! (end | (authenticated_*compo*)))U (end |
       (authenticated_*compo*) | (( ! (loginFail_*compo*)& ! (end |
       (authenticated_*compo*))) U (end | (authenticated_*compo*)|
       (((loginFail_*compo*) & ! (end | (authenticated_*compo*)))U (end |
       (authenticated_*compo*) | (( !((loginFail_*compo*)) & ! (end |
       (authenticated_*compo*))) U (end | (authenticated_*compo*) |
       (((loginFail_*compo*) & ! (end | (authenticated_*compo*))) U (end |
       (authenticated_*compo*) | (( ! (loginFail_*compo*) &! (end |
       (authenticated_*compo*))) U (end | (authenticated_*compo*)|
       (((loginFail_*compo*) & ! (end | (authenticated_*compo*))) U (end |
       (authenticated_*compo*) | ( ! (loginFail_*compo*) U (end |
       (authenticated_*compo*))))))))))))))))))))))))-> (! end U
       (lockout_*compo*))) U end);;;after 5 failed authentication attempts,
       the account is locked
5  GP-TM-26;;;G(passwordRecovery -> ! (blackListedWord)) ;;;password
       recovery system doesn't give too much information about existing users
6  GP-TM-38;;; G((sensitive_*sensitive*) -> encrypted) ;;;sensitive data are
       encrypted
7  GP-TM-42;;; G ((! (validResponse & (to_*compo*) & !
       (loginAttempt_*compo*)) U (authenticated_*compo*)) & (! (request &
       (to_*compo*) & ! (loginAttempt_*compo*)) U (authenticated_*compo*)))
       ;;;need authentication before receiving and sending data to an other
       component
8  GP-TM-48;;; G(((from_*dep*) & unavailable) -> ! (! output U (output &
       unavailable))) ;;;if an other component is unavailable, this
       component stay available
9  GP-TM-52(1);;;G(( request & (from_*compo*) & (XSS_*XSS*)) -> ( (! (
       response & (to_*compo*))) U (( response & (to_*compo*)) & (
       errorResponse | (! validResponse & response))))) ;;; the component
       doesn't accept requests that contains XSS
10 GP-TM-52(2);;;G(( request & (from_*compo*) &
       (SQLinjection_*SQLinjection*)) -> ( (! ( response & (to_*compo*))) U
       (( response & (to_*compo*)) & ( errorResponse | (! validResponse &
       response))))) ;;; the component doesn't accept requests that contains
       SQLinjection
11 GP-TM-53;;; G((errorResponse | ! validResponse) -> ! (blackListedWord))
       ;;; error messages doesn't contain too much information
```

Figure 7.8: Rules used by SMProVer.

```
rule "begin"
  when
  $t : Transition(source.isInit() == true)
  then
  $t.addParameter("begin=TRUE");
end

rule "end"
  when
  $t : Transition(target.isInit() == true)
  then
  $t.addParameter("end=TRUE");
end

rule "getUpdate"
  when
  $t1 : Transition(!isInput(), contain("get-update", "get_update") )
  $t2 : Transition(isInput(), contain("data"), getSource() ==
      $t1.getTarget())
  then
  $t2.addParameter("getUpdate_" + stringCleaner.clean($t2.getUpdate()) +
      "=TRUE");
  Main.keyWords.get("getUpdate_d").addValue(stringCleaner.clean($t2.getUpdate()));
end

rule "encrypted"
  when
  $t : Transition(isEncrypted())
  then
  $t.addParameter("encrypted=TRUE");
end

rule "searchUpdate"
  when
  $t : Transition(isReq() == true, isInput() == false,
      !contain2("updatefile", "findUpdate", "get_update").isEmpty())
  then
  for (String param: $t.contain2("update", "findUpdate", "get_update")){
    $t.addParameter("searchUpdate_"+ stringCleaner.clean(param) +
        "=TRUE");
    Main.keyWords.get("searchUpdate_d").addValue(stringCleaner.clean(param));
  }
end

rule "login"
  when
  $t1 : Transition(isReq() == true, contain("login", "user", "username"))
  $t2 : Transition(isReq() == false, isOk() == true, getSource() ==
      $t1.getTarget())
  $t3 : Transition(isReq() == true, contain("password", "pass", "pwd"),
      getSource() == $t2.getTarget())
  then
  $t1.addParameter("loginAttempt_" +
      stringCleaner.clean($t2.getOtherCompo()) + "=TRUE");
  $t2.addParameter("loginAttempt_" +
      stringCleaner.clean($t2.getOtherCompo()) + "=TRUE");
  $t3.addParameter("loginAttempt_" +
      stringCleaner.clean($t2.getOtherCompo()) + "=TRUE");
  Main.keyWords.get("loginAttempt_c").addValue(stringCleaner.clean($t2.getOtherCompo()));
end
```

Figure 7.9: Example of rules created by SMProVer, used by drools.

Bibliography

[1] Abbas Ahmad, Fabrice Bouquet, Elizabeta Fourneret, Franck Le Gall, and Bruno Legeard. Model-based testing as a service for iot platforms. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 727–742, Cham, 2016. Springer International Publishing.

[2] Bernhard K. Aichernig and Martin Tappler. 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*, pages 19–34, 2017.

[3] Rafa Al-Qutaish. Quality models in software engineering literature: An analytical and comparative study. *Journal of American Science*, 6, 11 2010.

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, January 2005.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, January 2002.

[6] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association.

[8] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer Berlin Heidelberg, 2006.

[9] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[10] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael 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*, ESEC/FSE '11, pages 267–277, New York, NY, USA, 2011. ACM.

[11] A.W. Biermann and J.A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597, June 1972.

[12] N. Chaabouni, M. Mosbah, A. Zemmari, C. Sauvignac, and P. Faruki. Network intrusion detection for iot security based on learning techniques. *IEEE Communications Surveys Tutorials*, 21(3):2671–2701, thirdquarter 2019.

[13] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the 2003 International Conference on Information Integration on the Web*, IIWEB'03, pages 73–78. AAAI Press, 2003.

[14] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: An interactive learning competition. In *Finite-State Methods and Natural Language Processing, 8th International Workshop, FSMNLP 2009, Pretoria, South Africa, July 21-24, 2009, Revised Selected Papers*, pages 139–146, 2009.

[15] Pierre Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, pages 222–237. Springer, 1996.

[16] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 411–420, May 1999.

[17] ENISA. Baseline security recommendations for iot in the context of critical information infrastructures, https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot, technical report, 2017.

[18] ENISA. Good practices for security of internet of things in the context of smart manufacturing , https://www.enisa.europa.eu/publications/good-practices-for-security-of-iot, technical report, 2018.

[19] ENISA. Enisa good practices for security of smart cars, https://www.enisa.europa.eu/publications/smart-cars, technical report, 2019.

[20] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[21] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158, 2009.

[22] Mengmeng Ge, Jin B. Hong, Walter Guttmann, and Dong Seong Kim. A framework for automating security analysis of the internet of things. *Journal of Network and Computer Applications*, 83:12 – 27, 2017.

[23] Roland Groz, Keqin Li, Alexandre Petrenko, and Muzammil Shahbaz. Modular system verification by inference, testing and reachability analysis. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, pages 216–233, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[24] L. Gutiérrez-Madroñal, I. Medina-Bulo, and J.J. Domínguez-Jiménez. Iot–teg: Test event generator system. *Journal of Systems and Software*, 137:784 – 803, 2018.

[25] Karim Hossen, Roland Groz, Catherine Oriat, and Jean-Luc Richier. 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*, pages 22–23, 2014.

[26] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin Heidelberg, 2012.

[27] Muhammad Naeem Irfan, Roland Groz, and Catherine Oriat. Improving Model Inference of Black Box Components having Large Input Test Set. In *ICGI 2012 - 11th International Conference on Grammatical Inference*, volume 21, pages 133–138, College Park, MD, United States, September 2012. Journal of Machine Learning Research. http://jmlr.org/proceedings/papers/v21/.

[28] Barnaby Jack. Broken hearts": How plausible was the homeland pacemaker hack. *IOActive, Seattle, Washington (blog. ioactive. com/2013/02/broken-hearts-how-plausible-was. html)*, 2013.

[29] Minhaj Ahmad Khan and Khaled Salah. Iot security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395 – 411, 2018.

[30] Gurjan Lally and Daniele Sgandurra. Towards a framework for testing the security of iot devices consistently. In Andrea Saracino and Paolo Mori, editors, *Emerging Technologies for Authorization and Authentication*, pages 88–102, Cham, 2018. Springer International Publishing.

[31] David Lo, Leonardo Mariani, and Mauro Santoro. Learning extended fsa from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063 – 2076, 2012. Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011).

[32] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[33] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 501–510, New York, NY, USA, 2008. ACM.

[34] 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*, 24(11):1921–1936, Nov 2012.

[35] Taras Maksymyuk, Stepan Dumych, Mykola Brych, Dimas Satria, and Minho Jo. An iot based monitoring framework for software defined 5g mobile networks. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM17, New York, NY, USA, 2017. Association for Computing Machinery.

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

[37] Leonardo Mariani and Mauro Pezze. Dynamic detection of cots component incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[38] Sara Nieves Matheu Garcia, José Hernández-Ramos, and Antonio Skarmeta. Toward a cybersecurity certification framework for the internet of things. *IEEE Security & Privacy*, 17:66–76, 05 2019.

[39] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on*

Program Comprehension, ICPC '18, pages 167–177, New York, NY, USA, 2018. ACM.

[40] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[41] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman. Iotsat: A formal framework for security analysis of the internet of things (iot). In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 180–188, Oct 2016.

[42] Ibrahim Nadir, Zafeer Ahmad, Haroon Mahmood, Ghalib Shah, Farrukh Shahzad, Muhammad Mujahid, Hassam Khan, and Usman Gulzar. An auditing framework for vulnerability analysis of iot system. pages 39–47, 06 2019.

[43] Mohd Nazir, Raees A. Khan, and Khurram Mustafa. A metrics based model for understandability quantification. *CoRR*, abs/1004.4463, 2010.

[44] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 19–30, New York, NY, USA, 2014. ACM.

[45] OWASP. Owasp testing guide v3.0 project, http://www.owasp.org/index.php/category: owasp _testing_project#owasp_testing_guide_v3. 2003.

[46] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. Timed k-tail: Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 401–411, 2017.

[47] Alexandre Petrenko and Florent Avellaneda. Learning communicating state machines. In Dirk Beyer and Chantal Keller, editors, *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*, volume 11823 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2019.

[48] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, pages 62–71, New York, NY, USA, 2005. ACM.

[49] Sébastien Salva and Elliott Blot. Model generation of component-based systems. *Software Quality Journal*, 28(2):789–819, January 2020.

[50] Sébastien Salva and Elliott Blot. Reverse engineering behavioural models of iot devices. In *31st International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Lisbon, Portugal, July 2019.

[51] Sandra Siby, Rajib Ranjan Maiti, and Nils Ole Tippenhauer. Iotscanner: Detecting and classifying privacy threats in iot neighborhoods. *CoRR*, abs/1701.05007, 2017.

[52] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[53] M. Tappler, B. K. Aichernig, and R. Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 276–287, March 2017.

[54] M. Tappler, B. K. Aichernig, and R. Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 276–287, March 2017.

[55] 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)*, pages 1–7, Nov 2015.

[56] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, pages 86–100, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[57] Moshe Vardi. Branching vs. linear time: Final showdown. volume 2031, pages 1–22, 03 2001.

[58] Judson Wilson, Riad Wahby, Henry Corrigan-Gibbs, Dan Boneh, Philip Levis, and Keith Winstein. Trust but verify: Auditing the secure internet of things. pages 464–474, 06 2017.

[59] K. P. Yoon and C.-L. Hwang. Multiple attribute decision making: An introduction (quantitative applications in the social sciences). 1995.

[60] Zhi-Kai Zhang, Michael Cheng Yi Cho, and Shiuhpyng Shieh. Emerging security threats and countermeasures in iot. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS15, pages 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and benchmarks for automated log parsing. *CoRR*, abs/1811.03509, 2018.