COnfECt : Une Méthode Pour Inférer Les Modèles De Composants D'un Système.

Elliott Blot, Sébastien Salva, Patrice Laurençot
LIMOS, Université Clermont Auvergne, France
Email: eblot@isima.fr, sebastien.salva@uca.fr, laurencot@isima.fr

Résumé

Cet article présente la méthode COnfECt qui complète les méthodes passives d'inférence de modèle pour extraire et représenter les comportements de composants d'un système vu comme une boite noire. COnfECt utilise pour cela les notions d'analyse de traces, de corrélation et de similarité de modèles. Nous montrons également comment intégrer COnfECt à la méthode GK-tail.

1 Introduction

L'inférence de modèles formels, aussi appelé *model learning*, regroupe un ensemble de méthodes permettant de retrouver un modèle comportemental d'un système, soit en interagissant avec ce dernier (méthodes actives), soit en analysant un ensemble de traces d'exécution obtenu en monitorant le système (méthode passive). Un modèle inféré peut ensuite être employé pour analyser le fonctionnement du système ou pour générer des tests. Bien qu'il soit aujourd'hui possible d'inférer des modèles à partir de certains systèmes réels, plusieurs points restent à étudier avant de pouvoir passer dans une phase industrielle. Parmi ceux-ci, nous avons relevé que les méthodes proposées considèrent un système comme une boite noire globale, qui prend des événements d'entrées depuis un environnement et produisent des événements de sorties. Pourtant, la grande majorité des systèmes actuels sont constitués de composants ré-utilisables, qui interagissent entre eux. La représentation de ces composants et de leurs compositions à travers plusieurs modèles permettrait une plus grande lisibilité du fonctionnement du système, voire une plus grande précision.

A travers cet article, nous nous penchons sur cette problématique et introduisons la méthode COnfECt (COrrelate Extract Compose) qui vise à compléter les méthodes d'inférence passives pour produire des systèmes de CEFSMs (Callable Extended FSM). Une CEFSM est une EFSM spécialisée équipée d'un événement interne spécial exprimant l'appel d'une autre CEFSM. L'idée fondamentale utilisée par COnfECt est qu'un composant dans un système peut être identifié par son comportement. D'une manière simplifiée, COnfECt analyse les traces, détecte les

blocs de comportements, extrait ces comportements dans des nouveaux ensembles de traces et aide à produire un ensemble de CEFSMs. COnfECt utilise pour cela les notions d'exploration d'événements (event mining), de corrélation d'événements et de clustering d'ensembles de traces.

Dans cet article, nous présentons sommairement les deux étapes de COnfECt appelées *Analyse de Trace et Extraction* et *Composition de CEFSMs*. Nous montrons enfin un exemple d'intégration de COnfECt avec la méthode GK-tail [11].

La suite de cet article est organisée de la manière suivante : la section 2 présente des travaux relatifs. La Section 3 présente les modèles générés par notre méthode, les CEFSMs. La description de l'approche est donnée dans la Section 4. Nous montrons un exemple d'intégration avec la méthode GK-tail, dans la Section 5 et nous concluons et donnons quelques perspectives en Section 6.

2 Travaux Relatifs

L'inférence de modèle est un ensemble de méthodes, permettant d'inférer un modèle à partir d'un système en analysant ses exécutions, et en les résumant dans un automate qui décrit le comportement du système [2]. Les modèles peuvent être générés à partir de différents types de données comme les réponses positives/négatives à des questions au systeme [3], des traces d'exécution [10], etc. On peut observer deux catégories de méthodes dans la littérature, les méthodes actives et les méthodes passives.

Les méthodes actives questionnent régulièrement le système ou l'utilisateur pour obtenir des observations négatives ou positives, qui seront étudiées pour en déduire le modèle. La plupart de ces méthodes ont été construites autour de deux concepts, l'algorithme $\mathcal{L}^*$ [3], et l'apprentissage incrémental [7]. L'algorithme $\mathcal{L}^*$ apprend un modèle de manière itérative, en questionnant un oracle, puis en recevant des contre-exemples, qui vont lui servir à améliorer ses hypothèses pour compléter son modèle, jusqu'à trouver le modèle final équivalent au système. Cet algorithme a été adapté à de nombreux systèmes et contextes [15, 1, 4, 9, 8]. Il a été optimisé pour réduire le nombre de questions posées à l'oracle. Cependant, avoir un oracle qui connaît tout du système est une hypothèse très forte. C'est pourquoi d'autres travaux proposent une approche différente appelée apprentissage incrémental [7]. Avec ce concept, des extraits négatifs ou positifs du système sont reçus successivement. Plusieurs modèles sont construits de manière à ce que si une nouvelle observation n'est pas consistante dans le modèle actuel, le prochain soit modifié en conséquence. Certaines de ces méthodes sont associées avec des méthodes de test active pour avoir une spécification initiales [14].

Les méthodes actives ne peuvent pas être appliquées à tous les systèmes. Par exemple, un système non-contrôlable ne peut pas être questionné facilement, et l'utilisation d'une méthode active pourrait provoquer des dysfonctionnements du système, car on a besoin de le redémarrer souvent. La catégorie des méthodes passives inclue les techniques qui infèrent un modèle à partir d'un ensemble d'ex-

traits, par exemple des traces. Étant donné qu'il n'y a pas d'interaction avec le système, ces méthodes sont appelées méthodes passives. Les modèles sont souvent construits en représentant ces extraits par un automate dont les états équivalents sont fusionnés ensuite. L'équivalence d'état se fait souvent au moyen d'abstraction de séquences d'événements ou d'abstraction d'états. Dans le cas d'abstraction de séquences d'événements, le niveau d'abstraction du modèle est augmenté en fusionnant les états qui on la même sequence d'événements. Ce processus est utilisé dans deux principaux algorithmes : Ktail [5] et kbehaviour [13]. Les deux algorithmes ont été ameliorés pour permettre de supporter les evenements combinés avec des données [12, 6], avec par exemple GKtail[11], une amélioration de Ktail.

L'approche que nous proposons a pour but de compléter les méthodes passives décrites précédemment pour détecter le comportement des composants d'un système, et inférer des CEFSMs correspondant. Les CEFSMs sont définis dans la prochaine section.

3 Callable Extended Finite State Machine

Une CEFSM permet de représenter l'appel d'un composant à partir d'un autre composant.

Définition 1 (CEFSM)**.** *un Callable Extended Finite State Machine (CEFSM) est un 5-tuple* $\langle S, S_0, \Sigma, P, T\rangle$ *où :*

- S *est un ensemble fini d'états,* $S_F \subseteq S$ *est l'ensemble non vide des états finaux,*
- S_0 *est l'état initial,*
- $\Sigma = \Sigma_I \cup \Sigma_O \cup \{call\}$ *est l'ensemble fini de symboles, avec* Σ_I *l'ensemble de symboles d'entrées,* Σ_O *l'ensemble de symboles de sorties, et* $call$ *une action interne,*
- P *un ensemble fini de paramètres, qui peuvent être assignés au valeurs de* D_P*,*
- T *est un ensemble fini de transitions. Une transition* $(s_1, e(p), G, s_2)$ *est un 4-tuple aussi noté* $s_1 \xrightarrow{e(p),G} s_2$ *où :*
  - $s_1, s_2 \in S$ *sont les états de la source et de la destination,*
  - $e(p)$ *est un événement avec* $e \in \Sigma$ *et* $p = \langle p_1, ..., p_k\rangle$ *un tuple fini de paramètre dans* $P^k (k \in \mathbb{N})$*,*
  - $G : D_P \rightarrow true, false$ *un prédicat qui restreint le franchissement des transitions.*

epareer Comme dit précédemment, une transition d'une CEFSM C_1 peut être annotée par un événement interne dénoté *call(CEFSM)* exprimant le début d'exécution d'une autre CEFSM. Cet événement non observable signifie que la CEFSM appelante C_1 est mise en pause tandis que la CEFSM appelée C_2 démarre son exécution à partir de son état initial. Quand C_2 a atteint un état final, C_1 reprend son

exécution après l'événement *call(CEFSM)*. Nous ne considérons pas qu'un composant retourne des résultats à un autre composant. Un système de CEFSMs est supposé inclure au moins une CEFSM qui appelle d'autres CEFSMs.

Nous disons également qu'une CEFSM C est callable-complete sur un système de CEFSMs SC, si toutes les CEFSMs de SC peuvent être appelées à partir de tous les états de C.

Définition 2 (CEFSM callable-complete)**.** *Soit* $SC = \{C_1, \ldots, C_k\}$ *un ensemble de CEFSMs.*

Un CEFSM $C = \langle S, S_0, \Sigma, P, T \rangle$ *est dit callable-complete sur* SC *ssi* $\forall s \in S, \exists s_2 \in S : s \xrightarrow{call(EFSM),G} s_2$*, avec* $G : \bigvee_{C' \in SC} CEFSM = C'$

4 Approche

COnfECt est une approche permettant de générer un système de CEFSMs à partir d'un système en boite noire, afin de compléter les méthodes d'inférence passive. Pour cela, COnfECt analyse les traces obtenues à partir du système à inférer et essaie de détecter les différents composants pour pouvoir les modéliser avec des CEFSMs. L'outil se base sur les traces obtenues via le système à inférer pour en extraire le comportement des composants. Par conséquent, plus le nombre de traces à disposition est importante, plus l'analyse de traces et l'inférence du modèle sera correcte.

Nous supposons que l'ensemble de traces est collecté de façon synchrone (environnement synchrone). Celles-ci peuvent être obtenues par des outils de monitoring. Dans ce papier, nous ne considérons pas le formatage des traces, et nous considérons que nous avons un "mapper" qui nous fournit des traces comme une séquence d'événements sous la forme $e(p_1 := d_1, ..., p_k := d_k)$ avec $p_1 := d_1, ..., p_k := d_k$ des affectations de paramètres.

COnfECt est composé de deux étapes principales appelées *Analyse de Traces et Extraction*, et *Composition de CEFSMs*, que nous allons décrire ci-dessous.

4.1 Analyse de Traces et Extraction

Après l'étude de plusieurs systèmes à base de composants, nous avons généralement observé qu'un composant produit un comportement souvent identifiable du reste des événements dans les différentes traces. De plus, certains composants, notamment dans des systèmes embarqués, produisent des événements non contrôlable (sortie non précédées par des entrées). A partir de ces observations, cette étape a pour but d'identifier les composants dans les traces à l'aide d'un *Coefficient de corrélation*, grâce à la détection des problèmes de contrôlabilité.

Le *coefficient de corrélation* nous permet d'évaluer le lien, ou relation entre deux événements successifs dans des traces. Nous définissons le *coefficient de cor-*

rélation entre deux événements par une fonction, dépendant de facteurs, qui sont définis par les préférences ou les besoins de l'utilisateur.

Définition 3 (Coefficient de corrélation)**.** *Soit* $e_1(p_1)$*,* $e_2(p_2)$ *deux évènements, et* $f_1(e_1(p_1), e_2(p_2)), ...f_k(e_1(p_1), e_2(p_2))$ *des facteurs de corrélation. Le coefficient de corrélation est défini par* $Corr(e_1(p_1), e_2(p_2)) = \sum_{i=1}^{k} f_i(e_1(p_1), e_2(p_2)).w_i$ *avec* $w_i \in \mathbb{R}_0^+$ *et* $\sum_{i=1}^{k} w_i = 1$

Un exemple de facteur est la fréquence d'apparition d'événements successifs dans les traces, ou encore la similitude entre les paramètres des événements. Ce coefficient est une valeur comprise entre 0 et 1, qui nous permet de déterminer les séquences d'événements dans une trace ayant une forte corrélation s'il est supérieur a un seuil X, ou s'ils ont une *faible corrélation* s'il est inférieur a un seuil Y. Les seuils X et Y dépendent du contexte, et doivent être déterminés par un expert.

Définition 4 (Faible et forte corrélation)**.** *Soit* $e_1(p_1)$ *et* $e_2(p_2)$ *deux événements dans une trace.*

- $e_1(p_1)$ *weak-corr* $e_2(p_2) \Leftrightarrow_{def} Corr(e_1(p_1), e_2(p_2)) < X$*. Nous disons que* $e_1(p_1)$ *est faiblement corrélé avec* $e_2(p_2)$*.*
- $e_1(p_1)$ *strong-corr* $e_2(p_2) \Leftrightarrow_{def} Corr(e_1(p_1), e_2(p_2)) > Y$*. Nous disons que* $e_1(p_1)$ *est fortement corrélé avec* $e_2(p_2)$

Les deux définitions précédentes sont spécialisées pour deux éléments. Nous formalisons la notion de *corrélation* au niveau des séquences d'événement.

Définition 5 (Corrélation entre séquences d'événements)**.** *Soit* $\sigma_1 = e_1(p_1)...e_k(p_k)$ *avec* $k > 1$*.*

- *match*$(\sigma_1) \Leftrightarrow_{def} \forall i \in [\![1\ ;\ k-1]\!] : e_i(p_i))$ *strong-corr* $e_{i+1}(p_{i+1})$
- σ_1 *mismatch* $\sigma_2 \Leftrightarrow_{def}$ *:*
  - $\sigma_2 = \epsilon$
    ou
  - $\sigma_2 = e'_1(p'_1)...e'_l(p'_l)$: *match*$(\sigma_1) \wedge e_k(p_k)$ *weak-corr* $e'_1(p'_1)$
    ou
  - $\sigma_2 = e'_1(p'_1)...e'_l(p'_l) : e'_1$ *est un symbole de sortie* $\wedge$(e_k *est un symbole de sortie* $\vee \sigma_1 = \epsilon$)

Le fonctionnement de l'algorithme *Analyse de Trace et Extraction* est donné dans l'Algorithme 1. Chaque trace σ est segmentée en sequence $\sigma'_1\sigma'_2...\sigma'_k$ (ligne 4, 5), grâce aux relations de corrélation entre séquences (*match* et *mismatch*). A chaque fois que nous avons σ'_i *mismatch* σ'_{i+1} entre deux séquences, nous considérons qu'il y a eu un appel à un autre composant. La trace σ est séparé en plusieurs séquences qui sont chacune placée dans une nouvelle trace, qui générera une nouvelle CEFSM.

La procédure *Extraire* prend la trace $\sigma = \sigma_1...\sigma_k$ et l'ensemble de traces T dans lequel σ sera ajouté quand sa transformation sera terminé. Pour une sequence σ_{id} de la trace σ, la procédure *Extraire* essaye de trouver une autre sequence σ_i

Algorithme 1 : Analyse de Trace et Extraction.

Entrée : $Traces = \{\sigma_1, \ldots, \sigma_m\}$
Sortie : $STraces = \{T_1, \ldots, T_n\}$

1 $T_1 = \{\}$;
2 $STraces = \{T_1\}$;
3 **pour chaque** $\sigma \in Traces$ **faire**
4 Trouver la sequence non vide $\sigma'_1\sigma'_2 \ldots \sigma'_k$ tel que : $\sigma = \sigma'_1\sigma'_2 \ldots \sigma'_k$, match$(\sigma'_i)_{(1 \leq i \leq k)}$, ($\sigma'_i$ mismatch $\sigma'_{i+1})_{(1 \leq i \leq k-1)}$;
5 Extraire($\sigma = \sigma'_1\sigma'_2 \ldots \sigma'_k, T_1$);
6 **Procédure** *Extraire*($\sigma = \sigma_1\sigma_2 \ldots \sigma_k, T$) **est**
7 $id := 1$;
8 **tant que** $id < k$ **faire**
9 $n := |STraces| + 1$;
10 $T_n := \{\}$;
11 $STraces := STraces \cup \{T_n\}$;
12 σ_p est le préfixe de σ jusqu'à σ_{id};
13 **si** $\exists i > id : match(\sigma_{id}\sigma_i)$ **alors**
14 σ_i est la première sequence dans $\sigma_{id} \ldots \sigma_k$ tel que match($\sigma_{id}\sigma_i$);
15 $\sigma := \sigma_p\sigma_{id}call(CEFSM := C_n)\sigma_i \ldots \sigma_k$;
16 **si** $(i - id) > 2$ **alors**
17 Extraire($\sigma_{id+1} \ldots \sigma_{i-1}, T_n$);
18 **sinon**
19 $T_n := T_n \cup \{\sigma_{id+1}\}$;
20 $id := i$;
21 **sinon**
22 $\sigma := \sigma_p\sigma_{id}call(CEFSM := C_n)$;
23 **si** $(k - id) > 1$ **alors**
24 Extraire($\sigma_{id+1} \ldots \sigma_k, T_n$);
25 **sinon**
26 $T_n := T_n \cup \{\sigma_k\}$;
27 $id := k$;
28 $T := T \cup \{\sigma\}$

tel que *match*($\sigma_{id}\sigma_i$) (ligne 14-16). La sequence $\sigma_{id+1}...\sigma_{i-1}$ (ou $\sigma_{id+1}...\sigma_k$ quand nous ne trouvons pas de σ_i correspondant) décrit le comportement d'un ou plusieurs composants. Si cette sequence est composée de plusieurs séquences, alors la procédure Extraire est utilisée récursivement sur celle-ci pour les séparer. Sinon la sequence est ajoutée à un nouvel ensemble de traces T_n, qui sera plus tard transformé en CEFSM Cn. Dans σ, la séquence $\sigma_{id+1}...\sigma_k$ est enlevée et remplacée par l'événement $call(CEFSM := C_n)$. Une fois que la procédure en a fini d'analyser σ, celui-ci est placé dans l'ensemble T.

Prenons l'exemple de trace σ donné en Figure 1 pour illustrer le fonctionnement de l'étape d'extraction de traces effectuée par la procédure Extraire. Celle-ci prend une trace déjà segmentée et un ensemble de traces T où sera stockée la séquence d'événements résultat. Au départ, nous appelons Extraire($\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6, T$).

A) L'algorithme commence avec σ_1. Supposons que la première séquence qui ait une forte corrélation avec σ_1 est σ_5. σ est transformé et devient $\sigma_1 call(CEFSM := C_2)\sigma_5\sigma_6$. $Extraire(\sigma_2\sigma_3\sigma_4, T_2)$ est récursivement appelé pour analyser $\sigma' = \sigma_2\sigma_3\sigma_4$.

B) $\sigma_2\sigma_4$ ayant une forte corrélation, σ' est modifié et devient $\sigma' = \sigma_2 call\ (CEFSM := C_3)\sigma_4$. La séquence σ_3 est une nouvelle trace du nouvel ensemble T_3. Maintenant que σ' est totalement parcourue, elle est ajoutée dans un ensemble de traces T_2.

C) Nous revenons à la trace σ, au niveau de la séquence σ_5 et réappliquons le même procédé. Comme il n'y a plus de séquence qui est fortement corrélée avec σ_5, la fin de la trace, correspond à un appel de composant. σ_6, est extraite et placée dans un nouvel ensemble de traces T_4. La trace σ devient $\sigma = \sigma_1 call(CEFSM := C_2)\sigma_5 call(CEFSM := C_4)$. Elle est placée dans l'ensemble de traces T_1.

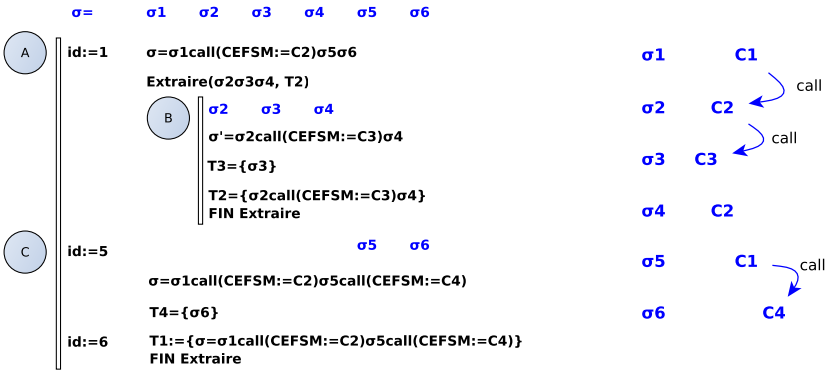

FIGURE 1 – Exemple d'execution de Extraire(σ, T).

A chaque fois que nous avons trouvé des comportements différents dans la trace, nous créons un nouvel ensemble de traces qui seront transformé en CEFSM.

4.2 Composition de CEFSMs

Le but de cette étape est d'analyser les compositions de CEFSMs obtenues. Nous proposons trois stratégies de composition dans ce papier. Ces stratégies sont réalisées sur les CEFSMs via l'Algorithme 2.

Composition stricte : (ligne (1,2) de l'Algorithme 2) Dans cette stratégie, nous cherchons à ne pas sur-généraliser le système de CEFSMs. Dans ce cas, une CEFSM ne peut être appelée qu'une fois et une seule dans le système de CEFSMs. La CEFSM appelée est alors composé d'un seul chemin. Cette stratégie de composition est déjà accomplie par l'étape *Analyse de Traces et Extraction*.

Composition faible : (ligne (3,14) de l'Algorithme 2) L'intuition de cette stratégie consiste à regrouper les CEFSMs ayant une forte similarité. En effet, les CEFSMs similaire doivent provenir du même composant. La notion de similarité est définie et évaluée par un coefficient de similarité. Ce coefficient est défini par le recouvrement (Overlap) des symboles et des paramètres entre 2 CEFSMs.

Définition 6. *Soit* $C_1 = \langle S_1, S0_1, \Sigma_1, P_1, T_1\rangle$ *et* $C_2 = \langle S_2, S0_2, \Sigma_2, P_2, T_2\rangle$ *deux CEFSMs. Le coefficient de similarité de* C_1 *et* C_2 *est definit par* $Similarité(C_1, C_2) = (Overlap(\Sigma_1, \Sigma_2) + Overlap(P_1, P_2))/2$

Algorithme 2 : CEFSM Composition

Entrée : CEFSM set SC, stratégie
Sortie : CEFSM set SC_f

1 **si** *stratégie = composition strict* **alors**
2 renvoyer SC;
3 **sinon**
4 $\forall (C_i, C_j) \in SC^2$ Calculer Similarité(C_i, C_j);
5 Construire une matrice de similarité;
6 Grouper les CEFSMs similaire dans des clusters $\{Cl_1, \dots Cl_k\}$;
7 **pour chaque** *cluster* $Cl = \{C_1, \dots, C_l\}$ **faire**
8 C_{Cl} :=Union disjointe des CEFSMs $C_1, \dots, C_l$;
9 $SC_f := SC_f \cup \{C_Cl\}$;
10 **pour chaque** $C = \langle S, S0, \Sigma, P, V, T \rangle \in SC_f$ **faire**
11 **pour chaque** $s_1 \xrightarrow{call(CEFSM),G} s_2 \in T$ *avec* $G : CEFSM = C_m$ **faire**
12 Trouver le Cluster Cl tel que $C_m \in Cl$;
13 Remplacer G par $G : CEFSM = C_{Cl}$;
14 Fusionner (s_1, s_2);
15 **si** *stratégie = composition forte* **alors**
16 **pour chaque** $C = \langle S, S0, \Sigma, P, T \rangle \in C_f$ **faire**
17 Compléter les états S avec une boucle C est callable-complete;
18 renvoyer SC_f

Nous construisons une matrice de similarité à partir des coefficients, ce qui nous permet d'utiliser de nombreux algorithmes de clustering pour trouver les classes de CEFSMs similaires, qui sont assemblées dans le même cluster. Les CEFSMs dans le même cluster sont fusionnées via une union disjointe. Toute les transitions $s_1 \xrightarrow{call(CEFSM:=C_i)} s_2$ sont mise à jour pour que la bonne CEFSM soit appelée, en remplaçant le C_i par la CEFSM correspondant au cluster. De plus, toutes les transitions $s_1 \xrightarrow{call(CEFSM:=C_i)} s_2$ sont remplacées par une boucle $(s_1, s_2) \xrightarrow{call(CEFSM:=C_i)} (s_1, s_2)$, en fusionnant les états s_1 et s_2.

Composition forte : (ligne (4,17) de l'Algorithme 2) Cette stratégie est basée sur la précédente, à laquelle nous rajoutons en plus pour tout état s, une transition $s \xrightarrow{call(CEFSM:=C_i)} s$ pour chaque CEFSMs C_i. Les CEFSMs deviennent maintenant callable-complete.

Dans la section suivante, nous montrons un exemple d'intégration avec la méthode passive GK-tail [11].

5 Intégration avec GK-tail

GK-tail est une méthode d'inférence de modèle passive proposée par Lorenzoli et al. [11], inférant une EFSM à partir de plusieurs traces d'exécution d'un système, et ceci en quatre étapes : la fusion de traces, la génération de prédicat, la construction d'une première EFSM, et enfin la fusion d'état.

Notre approche COnfECt peut s'intégrer à GK-tail comme décrit en Figure 2.

L'étape *Analyse de Traces et Extraction* de COnfECt permet de segmenter les traces et d'identifier des composants. Elle est réalisée après la fusion des traces (étape 1 GK-tail), pour accélérer le calcul du coefficient, car moins de traces sont à analyser. L'étape *Composition de CEFSMs* permet de grouper certains CEFSMs similaires, si la stratégie composition faible ou forte est employée. Nous faisons cette étape avant que GK-tail ne fusionne les états, car il est possible d'avoir plusieurs états similaires dans le même cluster, qui pourront être ensuite simplifiés dans la dernière étape, et donner un modèle plus léger.

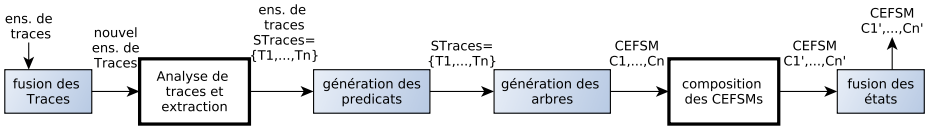

FIGURE 2 – Intégration des étapes avec GK-tail.

Pour comprendre le fonctionnement, prenons comme exemple un ensemble de traces composé en partie de la trace présentée en Figure 3.

```
e1:  ?/devices  (Verb=GET, Uri=/devices)
e2:  !/json.htm (Uri=/json.htm, type=command, param=udevice, idx=115, svalue=15.00)
e3:  ?Response (status=200, response=OK)
e4:  !Response (status=200, response=OK, data=[1])
e5:  !/json.htm (Uri=/json.htm, type=command, param=udevice, idx=115, svalue=16.00)
e6:  ?Response (status=200, response=OK)
e7:  ?/devices  (Verb=GET, Uri=/devices)
e8:  !Response (status=200, response=OK, data=[1])
e9:  ?/hardware  (Verb=GET, Uri=/hardware)
e10: !Response (status=200, response=OK, data=[2])
e11: ?/config  (Verb=GET, Uri=/config)
e12: !/json.htm (Uri=/json.htm, type=command, param=switchlight, idx=0, switchcmd=Off)
e13: ?Response (status=200, response=OK)
e14: !Response (status=200, response=OK, data=[3])
e15: ?/tools   (Verb=GET, Uri=/tools)
e16: !Response (status=200, response=OK, data=[4])
```

FIGURE 3 – Exemple d'un extrait de trace HTTP d'un objet connecté.

1) Nous exécutons la première étape de GK-tail, la fusion de traces, sur cette ensemble, qui reste inchangé n'étant composé que d'une seule trace.

2) Nous utilisons maintenant notre méthode *Analyse de Traces et Extraction*. Dans notre exemple nous utilisons deux facteurs de corrélation :

— $f_1(e_1(p_1), e_2(p_2)) = \frac{freq(e_1(p_1)e_2(p_2))}{freq(e_1(p_1))}$ avec $w_1 = 0,5$

— $f_2(e_1(p_1), e_2(p_2)) = \frac{freq(e_1(p_1)e_2(p_2))}{freq(e_2(p_1))}$ avec $w_2 = 0,5$

Avec $freq(e_1(p_1))$ (respectivement $e_2(p_2)$), la fréquence d'apparition de l'événement $e_1(p_1)$ (respectivement $e_2(p_2)$) dans toutes les traces, et $freq(e_1(p_1)e_2(p_2))$ la fréquence d'événements $e_1(p_1)$ qui ont $e_2(p_2)$ comme événement consécutif. Intuitivement, le facteur f_1 permet de déterminer si l'événement $e_1(p_1)$ cause généralement l'événement $e_2(p_2)$. Cependant, si tout les événements $e_2(p_2)$ se trouvent à la suite d'un événement $e_1(p_1)$ (qu'il y a donc une corrélation entre ces deux évé-

nements), et que celui-ci est en nombre beaucoup plus élevé, nous aurons un facteur faible. En effet, $freq(e_1(p_1)e_2(p_2)) \leq freq(e_2(p_2)$, et donc $freq(e_1(p_1)e_2(p_2))$ sera beaucoup plus faible que $freq(e_1(p_1))$. Ce cas arrive souvent dans notre exemple avec les événements de type *?Response* et *!Response* (qui représentent la moitié des événements). Donc nous rajoutons le facteur f_2 qui permet de déterminer si $e_2(p_2)$ est un conséquence possible de $e_1(p_1)$, et qui permet d'augmenter le coefficient dans ce genre de cas. La Figure 4 décrit l'ensemble de traces obtenu en utilisant l'algorithme 1, avec le coefficient de corrélation défini précédemment et les facteurs $X = 0,4$ et $Y = 0,4$.

```
Trace T1:
e1:  ?/devices  (Verb=GET, Uri=/devices)
     call (CEFSM=T2)
e4:  !Response (status=200, response=OK, data=[1])
     call (CEFSM=T3)

Trace T2:
e2:  !/json.htm (Uri=/json.htm, type=command, param=udevice, idx=115, svalue=15.00)
e3:  ?Response (status=200, response=OK)

Trace T3:
e5:  !/json.htm (Uri=/json.htm, type=command, param=udevice, idx=115, svalue=16.00)
e6:  ?Response (status=200, response=OK)
     call (CEFSM=T4)

Trace T4:
e7:  ?/devices  (Verb=GET, Uri=/devices)
e8:  !Response (status=200, response=OK, data=[1])
e9:  ?/hardware  (Verb=GET, Uri=/hardware)
e10: !Response (status=200, response=OK, data=[2])
e11: ?/config  (Verb=GET, Uri=/config)
     call (CEFSM=T5)
e14: !Response (status=200, response=OK, data=[3])
e15: ?/tools   (Verb=GET, Uri=/tools)
e16: !Response (status=200, response=OK, data=[4])

Trace T5:
e12: !/json.htm (Uri=/json.htm, type=command, param=switchlight, idx=0, switchcmd=Off)
e13: ?Response (status=200, response=OK)
```

FIGURE 4 – Exemple de traces après *Analyse de Traces et Extraction*. Chaque trace est dans un ensemble séparé.

3) Nous générons les prédicats associés aux paramètres à partir des valeurs en utilisant l'outil Daikon. Par exemple dans la Figure 4, les parties $svalue = 15.00$ et $svalue = 16.00$ deviennent $15.00 \leq svalue \leq 16.00$.

4) Nous construisons les premiers CEFSMs à partir des ensembles de traces obtenu dans l'étape précédente. Chaque ensemble de traces va donner un CEFSM en forme d'arbre. sur la Figure 5, on peut voir les CEFSMs générées, à partir des traces de la Figure 4.

5) Nous faisons la compositions des CEFSMs similaire. Dans cet exemple, nous avons choisi la composition faible. Pour cela nous calculons donc la matrice de similarité, en Figure 6, afin de les grouper en clusters, ici si la similarité est supérieure à $0,75$ (via une méthode de clustering hiérarchique agglomérative, ou AHC). Pour chaque cluster nous faisons ensuite l'union disjointe des CEFSMs. La

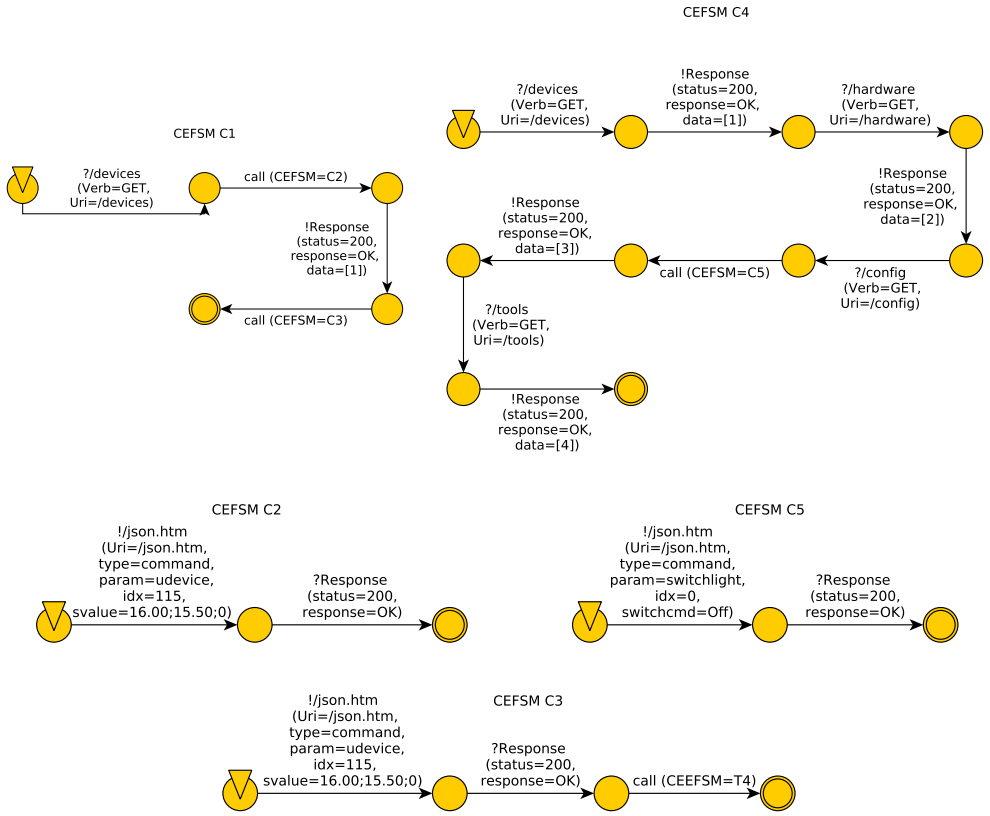

FIGURE 5 – Exemple de CEFSMs obtenues après la troisième étape de GKtail, a partir des traces.

	T1	T2	T3	T4	T5
T1	1	0,38	0,38	1	0,38
T2	0,38	1	1	0,38	0,8
T3	0,38	1	1	0,38	0,8
T4	1	0,38	0,38	1	0,38
T5	0,38	0,8	0,8	0,38	1

FIGURE 6 – Matrice de similarité.

Figure 7 représente les CEFSMs de la Figure 5, après avoir composé les CEFSMs similaires.

6) Pour finir nous terminons GK-tail en fusionnant les états similaire, pour obtenir le modèle final en Figure 8

La Figure 8 illustre un exemple de ce que nous pouvons obtenir en utilisant notre méthode avec GK-tail sur un petit échantillon de traces d'objet connecté. L'objet est composé d'une interface WEB, d'un capteur envoyant la température à

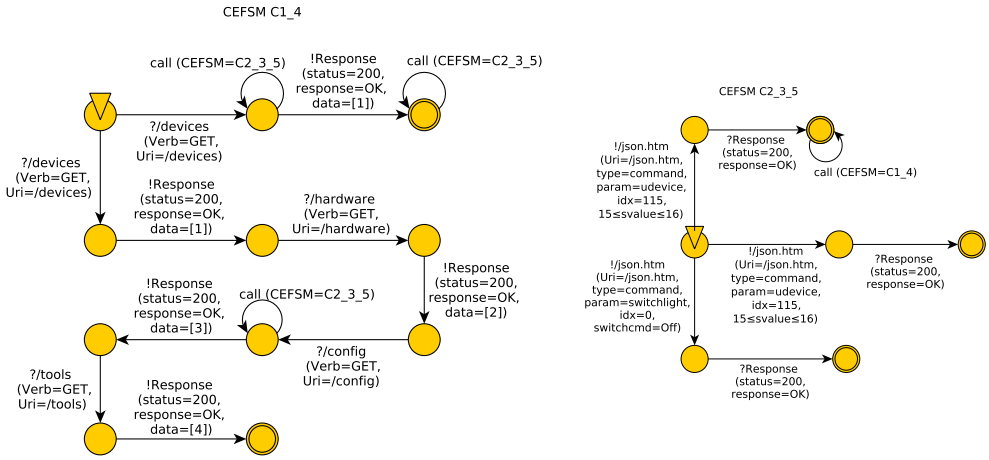

FIGURE 7 – Exemple de CEFSMs obtenues après l'étape *composition des CEFSMs*.

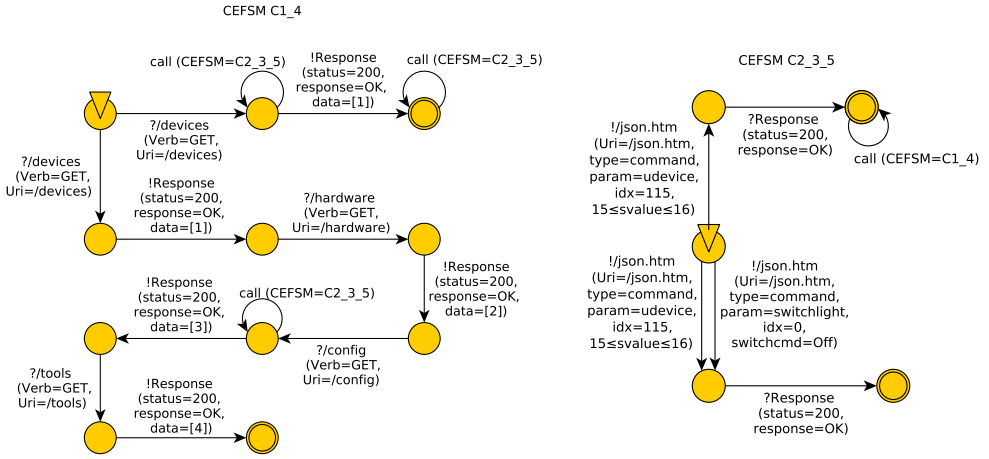

FIGURE 8 – Exemple de CEFSMs obtenues après fusion des états similaires.

un intervalle régulier, ainsi que d'un capteur de mouvement. Notre méthode permet de séparer d'un coté l'interface WEB, à gauche dans la Figure 8, et les capteurs à droite.

6 Conclusion

Nous avons introduit la méthode COnfECt, qui permet d'inférer un système de CEFSMs à partir de traces d'exécution d'une implémentation. Des composants sont identifiés dans les traces, et sont modélisés par des CEFSMs, qui peuvent être appelés à partir d'autres CEFSMs. En termes de perspectives, il faudrait adapter la méthode afin de l'utiliser sur des systèmes aux communications asynchrones, ou encore considérer des composants dont les exécutions se font en parallèle. Nous

prévoyons l'implémentation de cette méthode et son intégration avec GK-tail pour effectuer des évaluations.

Références

[1] R. ALUR, P. ČERNÝ, P. MADHUSUDAN et W. NAM : Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, jan. 2005.

[2] G. AMMONS, R. BODÍK et J. R. LARUS : Mining specifications. *SIGPLAN Not.*, 37(1):4–16, jan. 2002.

[3] D. ANGLUIN : Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[4] T. BERG, B. JONSSON et H. RAFFELT : Regular inference for state machines with parameters. *In* L. BARESI et R. HECKEL, éds : *Fundamental Approaches to Software Engineering*, vol. 3922 de *Lecture Notes in Computer Science*, p. 107–121. Springer Berlin Heidelberg, 2006.

[5] A. BIERMANN et J. FELDMAN : On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6): 592–597, June 1972.

[6] D. COTRONEO, R. PIETRANTUONO, L. MARIANI et F. PASTORE : Investigation of failure causes in workload-driven reliability testing. *In Fourth international workshop on Software quality assurance : in conjunction with the 6th ESEC/FSE joint meeting*, p. 78–85. ACM, 2007.

[7] P. DUPONT : Incremental regular inference. *In Proceedings of the Third ICGI-96*, p. 222–237. Springer, 1996.

[8] K. HOSSEN, R. GROZ, C. ORIAT et J. 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*, p. 22–23, 2014.

[9] F. HOWAR, B. STEFFEN, B. JONSSON et S. CASSEL : Inferring canonical register automata. *In* V. KUNCAK et A. RYBALCHENKO, éds : *Verification, Model Checking, and Abstract Interpretation*, vol. 7148 de *Lecture Notes in Computer Science*, p. 251–266. Springer Berlin Heidelberg, 2012.

[10] I. KRKA, Y. BRUN, D. POPESCU, J. GARCIA et N. MEDVIDOVIC : Using dynamic execution traces and program invariants to enhance behavioral model inference. *In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, p. 179–182, New York, NY, USA, 2010. ACM.

[11] D. LORENZOLI, L. MARIANI et M. PEZZÈ : Automatic generation of software behavioral models. *In Proceedings of the 30th International Conference*

on Software Engineering, ICSE '08, p. 501–510, New York, NY, USA, 2008. ACM.

[12] L. MARIANI et F. PASTORE : Automated identification of failure causes in system logs. *In Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, p. 117–126, Nov 2008.

[13] L. MARIANI et M. PEZZE : Dynamic detection of cots component incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[14] K. MEINKE et M. SINDHU : Incremental learning-based testing for reactive systems. *In* M. GOGOLLA et B. WOLFF, éds : *Tests and Proofs*, vol. 6706 de *Lecture Notes in Computer Science*, p. 134–151. Springer Berlin Heidelberg, 2011.

[15] H. RAFFELT, B. STEFFEN et T. 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, p. 62–71, New York, NY, USA, 2005. ACM.