

Une approche incrémentale combinant test et extraction de modèles

Roland Groz², Muzammil Shahbaz¹, and Keqin Li²

¹ France Telecom R&D
Meylan, France.

`muhammad.muzammilshahbaz@orange-ftgroup.com`

² LIG, Computer Science Lab
Grenoble Universités, France

`{Keqin.Li,Roland.Groz}@imag.fr`

Un des principaux obstacles à l'utilisation d'approches formelles est l'absence de modèles. Nous nous intéressons ici au problème de l'intégration et du test de composants logiciels. Dans la pratique industrielle actuelle, il est de plus en plus fréquent de travailler en intégrant des composants logiciels externes dont aucun modèle n'est disponible. La documentation fournit en général une spécification insuffisante.

Afin de guider l'intégration, nous proposons d'utiliser des algorithmes d'inférence de machines pour extraire des modèles des composants. L'interaction entre ces modèles dans une composition permet de déduire d'autres tests qui peuvent être confrontés au système. L'ingénieur chargé de l'intégration dispose ainsi d'outils lui permettant de découvrir les interactions effectives entre les composants et de guider son processus d'intégration et de test pour valider les combinaisons de comportements. Nous travaillons sur des modèles d'automates étendus avec des paramètres et des prédicats, mais sans variables internes.

1 Introduction

Alors que le développement logiciel était pendant longtemps une activité intégrée au sein d'une entreprise qui fabriquait l'ensemble de ses produits logiciels, on procède de plus en plus par assemblage de composants provenant de sources externes.

Cette évolution implique de nouveaux défis pour intégrer les approches formelles dans le développement logiciel du côté de l'intégrateur. D'abord, les composants externes sont rarement accompagnés de spécifications formelles. Plus largement, on est confronté à l'absence ou à la non-disponibilité des modèles ou des éléments qui ont permis de construire ces composants. En général, on dispose d'une documentation qui n'est pas forcément suffisamment précise pour répondre aux questions que se pose l'intégrateur, et qui ne peut pas être intégrée directement dans les processus de développement de l'assemblage.

Nous proposons une approche formelle assistant l'intégrateur de composants dans sa tâche d'expérimentation et de test de composants dans une architecture définie a priori. Nous voulons permettre une approche de *test* basée sur

des *modèles* (avec génération automatique de tests), même en l'absence de modèles initiaux pour les composants. Pour cela, nous combinons une approche descendante de génération de tests (pour l'assemblage) à partir de modèles avec une approche ascendante de reconstruction de modèles (de chaque composant) à partir des observations issues du test.

L'objectif est de permettre un test suffisant pour mettre en évidence les interactions des composants dans les divers cas d'usage de l'assemblage. On ne cherche donc pas à dériver un modèle complet des composants, mais une *approximation* en phase avec l'objectif. En général, dans un assemblage, seule une petite partie du fonctionnement d'un composant est sollicitée. Comme le modèle correspondant à cette partie sera dérivé des tests, la génération de tests sur un seul composant ne serait pas pertinente pour découvrir des erreurs (puisque les tests seraient déduits du composant lui-même, donc corrects par construction). C'est pourquoi notre approche prend tout son sens dans un contexte *d'intégration*, où les modèles permettent de déduire de nouvelles interactions à tester. En outre, l'expérience montre que les problèmes d'interfonctionnement entre composants proviennent souvent du traitement de certaines valeurs échangées, alors qu'en général les interfaces sont cohérentes puisqu'elles ont bien été étudiées dans la définition de l'architecture d'intégration. C'est pourquoi nous accordons une importance particulière à l'extraction de modèles paramétrés : chaque interaction entre composants porte également des valeurs.

1.1 Une approche globale

Nous travaillons sur une représentation des composants par des automates communicants. La communication se fait par des symboles d'entrées-sorties, représentant un type d'interaction, enrichis par des paramètres représentant les valeurs échangées lors de l'interaction. Les symboles correspondent par exemple aux primitives décrites dans la documentation du composant.

Nous faisons les hypothèses suivantes.

- Les composants sont des *boîtes noires*. On connaît leurs interfaces, c'est à dire les types des entrées, et accessoirement de leurs sorties. Il est possible qu'on n'en connaisse qu'un sous-ensemble. On connaît par exemple les primitives de services offertes par le composant et le type de leurs paramètres. C'est ce dont disposerait un ingénieur avec la documentation du composant.
- Nous nous plaçons dans une hypothèse de fonctionnement *réactif* du système. Les entrées globales ne seront fournies au système que dans un état stable de celui-ci, où il n'y a plus d'interaction interne possible. On supposera aussi que même si la communication entre composants du système est asynchrone, il n'y a qu'un flot de traitement actif (un seul message en transit dans le système).
- On dispose de *scénarios d'usage* du système permettant d'avoir un ensemble de séquences d'entrées et des valeurs significatives des paramètres associés. L'ingénieur qui a conçu l'architecture pour composer un service à partir des composants aura aussi élaboré des scénarios d'usage typiques,

et il s’agit, maintenant que l’architecture est définie, de voir si les composants interagissent bien selon les attentes qu’on en a. Un apport clé de notre approche va être de permettre l’enrichissement des tests, et la déduction automatique de tests systématiques à partir d’un nombre restreint de scénarios de test.

- Les composants sont intégrés, mais peuvent être aussi testés isolément. Lorsqu’ils sont intégrés, certaines de leurs interfaces sont connectées entre elles (interfaces internes), d’autres restent ouvertes pour communiquer avec l’environnement. On suppose que les interfaces internes restent *observables* (mais pas forcément contrôlables).
- Accessoirement, il se peut qu’on dispose de modélisations très partielles, sous forme d’automates à entrées et sorties (paramétrées), de certains composants, représentant par exemple leur flot de contrôle pour certains scénarios. Notre approche saurait intégrer de tels modèles, mais en leur absence, on peut reconstruire des modèles à partir de rien.

Notre approche va consister à tester d’abord chaque composant selon un *algorithme d’inférence d’automate* qui permettra d’en dériver un premier modèle cohérent avec les observations sur un nombre restreint d’entrées sorties correspondant à celles des scénarios d’usage fournis initialement. Ensuite, nous assemblerons les modèles, et dériverons des tests d’intégration nous permettant de confronter le fonctionnement du système réel aux modèles préliminaires. Ceci nous permettra d’enrichir ces modèles et de déduire de nouveaux tests correspondants aux cas “croisés”, d’interactions entre composants qui n’avaient pas été testées dans les scénarios antérieurs. Nous avons donc une approche incrémentale et itérative alternant les phases de test avec les phases d’enrichissement des modèles. Le processus peut être arrêté lorsqu’on ne détecte plus d’incohérence entre les modèles et le système ou lorsqu’on a atteint un certain objectif de couverture du système.

Dans la suite de cet article, nous présentons d’abord un exemple simple permettant d’illustrer les algorithmes d’inférence d’automates. Ensuite, nous illustrons l’algorithme lorsqu’on se restreint à des automates sans paramètres (machines de Mealy). Nous étendons le modèle pour prendre en compte les paramètres et expliquons comment l’algorithme peut être étendu. Enfin, nous présentons la méthode d’intégration qui s’appuie sur les algorithmes d’inférence.

2 Un exemple de composant à identifier

Nous donnons un exemple de contrôleur de climatiseur qui régule un système de chauffage et de ventilation. La structure interne du contrôleur et le fonctionnement du système encapsulé par ce composant ne sont pas connus. Il s’agit donc d’un composant considéré comme une boîte noire. Par contre, on peut connaître un ensemble d’entrées en analysant ses interfaces externes. La figure 1 présente un diagramme global du contrôleur.

Le contrôleur de climatiseur accepte des entrées venant de l’environnement pour contrôler le système. Il reçoit les signaux *Marche* et *Arrêt* pour allumer

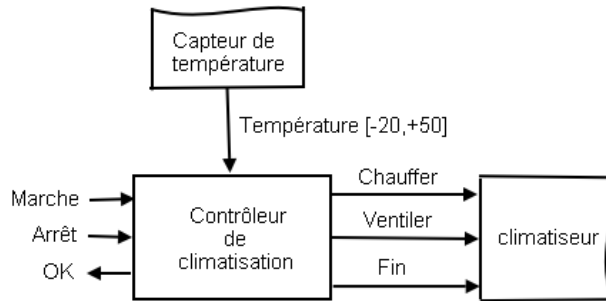


FIG. 1. Un diagramme global de climatiser

et éteindre le système et Température T qui change le mode du système. Il accepte des valeurs de température entre -20 Celsius et $+50$ Celsius, qui sont des paramètres de l'entrée T . Ensuite, il donne une commande *Chauffer* pour mettre en route le chauffage s'il fait froid (entre -20°C et 11°C) ou *Ventiler* pour la ventilation s'il fait chaud (entre 16°C et 50°C). La commande de sortie porte des paramètres pour régler la vitesse de système, par exemple, s'il ne fait pas très froid, le contrôleur envoie une commande pour ralentir le chauffage. De même, il envoie une commande pour relancer la climatisation s'il fait très chaud, sinon, il peut suffire de ventiler.

Cet exemple classique va nous servir à illustrer le fonctionnement des algorithmes d'inférence de modèles. Dans un premier temps, nous montrerons comment on peut en apprendre un modèle d'automate simple, en faisant une abstraction des paramètres. Puis nous étendons l'algorithme pour en apprendre un modèle paramétré.

3 Apprentissage d'automates simples

Le modèle d'automate que nous considérerons sera celui des automates à entrées et sorties (appelés aussi machines de Mealy). Un automate est un sextuplet $M = (Q, I, O, \delta, \lambda, q_0)$, où Q est l'ensemble des états, $q_0 \in Q$ est l'état initial, I l'alphabet d'entrée, O celui des sorties, $\delta : Q \times I \rightarrow Q$ est la fonction de transition d'états et $\lambda : Q \times I \rightarrow O$ est la fonction de sortie. Les ensembles Q, I, O sont bien sûr finis. On s'intéressera à des automates complètement définis, c'est à dire tels que $\text{dom}(\delta) = \text{dom}(\lambda) = Q \times I$. Au besoin, on complètera pour cela l'automate avec un symbole supplémentaire Ω qui sera une abstraction pour la réponse d'un système à des entrées non valides dans un état donné, sur lequel on rajoutera une boucle étiquetée par Ω en sortie.

Nous cherchons à déduire la structure de contrôle d'un composant qu'on peut tester en boîte noire. Pour cela, nous nous sommes intéressés aux travaux similaires menés soit dans le domaine de l'ingénierie à partir de scénarios [13] [15] [4], soit en vérification [5] [6] ou en test [8] [9]. Dans ce domaine, l'algorithme de base le plus efficace dans ce contexte de test actif est l'algorithme d'Angluin [1]. C'est

un algorithme de complexité polynomiale (en nombre de requêtes au système boîte noire) conçu pour des automates accepteurs déterministes. Notre travail actuel sur l'inférence de machines se situe dans la lignée d'adaptations et d'extensions à l'algorithme d'Angluin pour des modèles comportant des entrées et des sorties avec des paramètres et des prédicats.

Dans la plupart des travaux antérieurs, l'algorithme d'Angluin a été utilisé sans modification avec une simple correspondance entre les symboles d'entrées et de sortie et l'alphabet A d'un automate accepteur. Par exemple, en prenant $A = I \cup O$ [7] [9], ou en prenant des couples (entrée, sortie) $A = I \times O$ [13] comme lettres de l'alphabet. Nous avons proposé une adaptation de l'algorithme dans laquelle on remplace la notion d'acceptation (décision binaire, codée par 0 et 1 dans l'algorithme d'Angluin) par les sorties correspondant aux derniers symboles entrés [11]. Par ailleurs, l'algorithme d'Angluin a été initialement proposé dans le contexte d'un apprentissage dans lequel un oracle connaissant le contenu de la boîte noire peut être interrogé de deux façons.

- par une requête d'appartenance, qui permet de savoir si une séquence d'entrées est un mot accepté (reconnu) par l'automate; dans notre approche, c'est le composant qui servira d'oracle et fournira les sorties correspondant à la séquence d'entrée
- par une requête d'équivalence, qui permet de savoir si l'automate minimal cohérent avec les observations enregistrées qu'on a pu reconstruire est équivalent à l'automate caché dans la boîte noire; s'il ne l'est pas, l'oracle fournit un contre-exemple, i.e. une séquence reconnue par la boîte noire mais pas par l'automate construit; dans notre approche, il n'y a pas de tel oracle omniscient, c'est le test d'intégration qui fournira un éventuel contre-exemple, ou le critère d'arrêt du test qui terminera l'algorithme sur un modèle approché.

Noter qu'on supposera toujours que le composant est réinitialisable ce qui permet de reprendre toutes les séquences de test à partir de l'état initial.

3.1 Algorithme pour machine de Mealy

Nous ne décrivons pas ici le détail de l'algorithme qui a été présenté ailleurs [11], mais nous en donnons les principes et une illustration sur l'exemple du climatiseur. Conformément à l'algorithme d'Angluin, on note les observations faites au cours du test dans une table appelée table d'observation, qui sera étendue progressivement au fur et à mesure des observations. Nous notons cette table (S, E, TO) . Les lignes de cette table sont étiquetées par des séquences d'entrées, qui correspondent à des préfixes pour atteindre les états de l'automate. Soit S l'ensemble de ces séquences. On impose à S d'être clos par préfixe. Les colonnes sont elles aussi étiquetées par des séquences, d'un ensemble E auquel on imposera d'être clos par suffixe.

A l'intersection d'une ligne s de S et d'une colonne e de E on portera dans la table la valeur $TO(s, e)$ qui sera la séquence des sorties produites par le composant en réponse à la séquence e , lorsqu'il a été préalablement positionné dans un état par la séquence s .

On commence l'algorithme avec $S = \{\epsilon\}$ (le mot vide) et $E = I$ (l'ensemble des symboles d'entrées du composant). On remplit la première (et seule) ligne avec les résultats des tests consistant à appliquer chacune des entrées dans l'état initial : dans chaque case, on trouve donc la (ou les) sortie(s) correspondant à cette entrée (celle de la colonne de cette case).

On vérifie ensuite la "complétude" de la table en rajoutant toutes les lignes correspondant à $S \cdot I$. A tout moment, dans l'algorithme, il y aura ainsi deux parties dans la table : la partie haute, des lignes étiquetées par S et la partie basse qui étend celles-ci d'une entrée, pour toutes les entrées possibles. Un exemple de table d'observation est donné en figure 2.

	E		
	e1	e2	
S	ε	o1	o2
S•I	s1	o1	o3
	s2	o2	o1

FIG. 2. Structure d'une table d'observation

La table sera dite *close* si toute ligne t de $S \cdot I$ est le doublon d'une ligne de S , i.e; s'il existe s de S tel que $ligne(s) = ligne(t)$. Elle sera dite *cohérente* si lorsque deux éléments de S , s_1 et s_2 ont des lignes identiques, alors pour toute entrée a de I , $ligne(s_1 \cdot a) = ligne(s_2 \cdot a)$.

Lorsqu'on arrive à un état où la table est *close* et *cohérente*, comme illustré sur la table 1, alors on construit un modèle minimal d'automate (la conjecture) qui permet d'expliquer toutes ces observations de la façon suivante.

- $Q = \{ligne(s) : s \in S\}$
- $q_0 = ligne(\epsilon)$
- $\delta(ligne(s), a) = \{ligne(s \cdot a), a \in I\}$
- $\lambda(ligne(s), a) = \{TO(s \cdot a), a \in I\}$

Chaque séquence préfixe $s \in S$ est un chemin d'accès qu'on associe à un état. Deux séquences peuvent conduire au même état, et la propriété de cohérence garantit qu'alors les observations ultérieures restent cohérentes (pour toutes les entrées et suffixes discriminants déjà recensés). La propriété de clôture garantit qu'on n'atteint pas de nouveaux états, différents des précédents, en prolongeant la séquence. Il est assez facile de montrer que l'automate construit est compatible avec la table d'observation si elle est *close* et *cohérente*, c'est à dire que pour

tout s de $S \cup S \cdot I$ et pour tout e de E , $\lambda(q_0, s \cdot e) = \lambda(q_0, s) \cdot TO(s \cdot e)$ (en ayant étendu λ aux séquences d'entrée). On peut par ailleurs montrer comme dans [1] que l'automate ainsi obtenu est l'automate minimal compatible avec la table d'observation.

Lorsque la table n'est pas *close*, cela signifie qu'on a découvert une nouvelle séquence du type $s' = s \cdot a$ menant à un état inéquivalent aux états précédemment identifiés. On rajoute alors cette séquence s' à S (la ligne correspondante passe de la partie basse à la partie haute de la table), et on complète la table en partie basse par les lignes correspondant à $s' \cdot I$. Lorsque la table n'est pas *cohérente*, on a trouvé deux éléments s_1 et s_2 tels que $ligne(s_1) = ligne(s_2)$ alors qu'il existe a de I et e de E tels que $ligne(s_1 \cdot a \cdot e) \neq ligne(s_2 \cdot a \cdot e)$. Ce qui signifie que $a \cdot e$ est une séquence discriminant les deux états atteints par s_1 et s_2 . On rajoute $a \cdot e$ à E et on complète la table (entre autres, pour vérifier si cette séquence pourrait discriminer d'autres préfixes). On continue ainsi jusqu'à obtenir une table *close* et *cohérente*, qui permet de construire un automate conjecturé. Les préfixes présents dans S représentent les chemins d'accès à des états "candidats". Les suffixes présents dans E représentent les séquences dont on a pu établir jusqu'à ce stade qu'elles permettent de distinguer les états 2 à 2.

Une conjecture, qui est compatible avec toutes les observations faites jusqu'à son établissement, peut être remise en cause si des observations ultérieures ne sont pas conformes aux prédictions de l'automate. Dans l'algorithme original d'Angluin, c'est un expert qui fournit un contre-exemple. Dans notre approche, le contre-exemple viendra d'un test ultérieur dans une phase d'intégration. On va alors enrichir la table en étendant S avec le contre-exemple et tous ses préfixes (non déjà présents dans la table). On complète alors la table avec les observations requises jusqu'à aboutir à une nouvelle conjecture.

L'intérêt de l'algorithme est que si le composant a n états, l'algorithme terminera en un temps polynomial en n . Il en sera de même si le composant peut être abstrait en un système à n états en ne distinguant pas les valeurs des paramètres.

On peut noter que l'algorithme est incrémental puisque l'insertion d'un contre-exemple ne fait que rajouter des lignes à la table précédemment construite. Cette caractéristique est également utile si on dispose d'un modèle initial du composant sous forme d'un automate. On peut associer à cet automate une table pour laquelle S est construit à partir des plus courtes séquences d'accès à chaque état et $E = I$. Il faut bien sûr que l'automate fourni initialement soit conforme au fonctionnement du système au moins sur les séquences de $S \cup S \cdot I \cdot I$.

3.2 Apprentissage du contrôleur de climatiseur comme machine de Mealy

Dans une machine de Mealy, il ne peut y avoir de paramètres sur les symboles d'entrée et de sortie. On pourrait certes associer un symbole différent à chaque valeur du paramètre. Par exemple, $T(12)$ pourrait être un symbole d'entrée qu'on pourrait noter $T12$. Mais ceci aurait plusieurs inconvénients.

- On devrait discrétiser l'ensemble du domaine des paramètres d'entrée. Pour les températures de notre exemple (de -20°C à $+50^\circ\text{C}$), il y aurait 71 valeurs,

	M	A	BT	MT	HT
ϵ	OK	Ω	Ω	Ω	Ω
M	Ω	Ω	C	F	V
$M - BT$	Ω	F	C	F	Ω
$M - HT$	Ω	F	Ω	F	V
$M - MT$	Ω	Ω	C	F	V
$M - BT - A$	OK	Ω	Ω	Ω	Ω
$M - BT - BT$	Ω	F	C	F	Ω
$M - BT - MT$	Ω	Ω	C	F	V
$M - HT - A$	OK	Ω	Ω	Ω	Ω
$M - HT - MT$	Ω	Ω	C	F	V
$M - HT - HT$	Ω	F	Ω	F	V

TAB. 1. Une table *close* et *cohérente* pour apprendre le contrôleur de climatiseur

si on s'en tient à des valeurs entières, or il pourrait être souhaitable d'avoir un grain plus fin. Pour des valeurs flottantes, on pourrait donc avoir un grand nombre de valeurs.

- Ceci conduirait à une complexité inutile de l'automate, rédhitoire pour l'apprentissage et l'utilisation.
- On perdrait la structure de contrôle de l'automate, en la mélangeant avec les données. Or l'intérêt de la modélisation est de pouvoir extraire le contrôle pour en déduire une abstraction représentative et pertinente pour l'utilisation de techniques de génération de tests.

Pour apprendre un modèle de Mealy du contrôleur de climatiseur, nous avons besoin de réarranger son ensemble d'entrée pour que l'algorithme puisse être utilisé commodément. Ceci ne donnera pas un fonctionnement détaillé du contrôleur mais une abstraction de son comportement face à un changement de température.

Par exemple, nous distinguons trois sous-domaines de température, auxquels nous associons donc 3 entrées. Nous remplaçons T par MT (la température moyenne, entre 12°C et 15°C), BT (température basse, entre -20°C et 11°C) et HT (température haute, entre 16°C et 50°C). Donc, nous construisons $I = \{M, A, BT, MT, HT\}$ pour l'algorithme de machine de Mealy. La table 1 est la table d'observation obtenue en fin d'exécution de l'algorithme, quand la table est *close* et *cohérente*. La figure 3 est la conjecture pour le contrôleur comme machine de Mealy déduite de la table. Pour simplifier, les lignes qui ne correspondent à aucune sortie ne sont pas affichées dans la table. Aussi, nous utilisons les abréviations à la place des noms complets des entrées et sorties dans la table et la figure : M (Marche), A (Arrête), C (Chauffer), V (Ventiler), F (Fin).

Il faut noter que plus on raffiner les domaines d'entrée, plus on augmentera la taille de la machine de Mealy calculée, il faut donc éviter une explosion du nombre d'états.

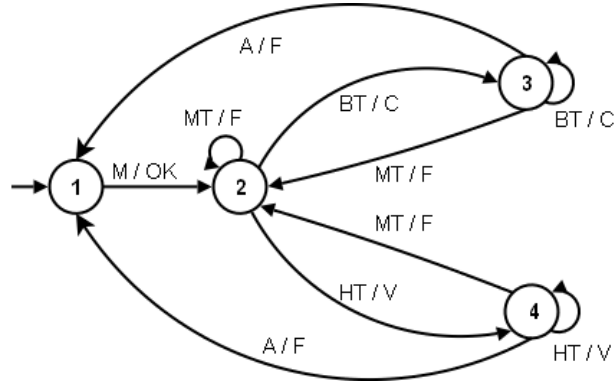


FIG. 3. Conjecture de Mealy

4 Modèle d'automates paramétrés

Comme on l'a noté, le modèle d'automates sans paramètres nécessite des abstractions de modélisation délicates. En outre, dans le contexte du test, il présente l'inconvénient majeur de ne pas instancier les valeurs. Les modèles d'automates obtenus pour les composants ne permettraient donc pas de produire des séquences de test complètes. C'est pourquoi nous proposons d'étendre le modèle pris en compte à des automates dans lesquels les entrées et les sorties sont paramétrés. Afin de prendre en compte les calculs différents auxquels peuvent conduire des valeurs différentes des paramètres pour une même entrée, on considèrera également que les transitions de l'automate peuvent être accompagnées d'une garde (prédicat) portant sur les valeurs des paramètres d'entrée. En cela, nous étendons le modèle paramétré que nous avons proposé dans [12]

En revanche, nous considérons des automates sans variables internes. En effet, autant on peut observer les valeurs des entrées et sorties de la boîte noire, autant on n'a pas accès aux états internes et à la structure de la machine. Il n'est donc pas possible de distinguer entre une mémorisation par des structures de contrôle ou dans des mémoires. Nous discutons cette restriction plus bas.

Le modèle que nous proposons est donc le suivant. Un automate paramétré (PFSM) est un septuplet : $M = (Q, I, O, D_I, D_O, T, q_0)$.

- Q est un ensemble fini d'états
- $q_0 \in Q$ est l'état initial.
- I est un ensemble fini de symboles d'entrée.
- O est un ensemble fini de symboles de sortie.
- D_I est un ensemble de valeurs du paramètre d'entrée.
- D_O est un ensemble de valeurs du paramètre de sortie.
- T est l'ensemble des transitions.

Une transition $t \in T$ est décrite par : $t = (q, q', i, o, p, f)$, où $q \in Q$ est l'état de départ, $q' \in Q$ est l'état d'arrivée, $i \in I$ le symbole d'entrée, $o \in O$ celui de sortie, $p \subseteq D_I$ est un prédicat sur la valeur du paramètre et $f : p \rightarrow D_O$ est la

fonction de calcul du paramètre de sortie pour cette transition : elle associe la valeur du paramètre de sortie correspondant à celle du paramètre d'entrée.

Noter que les ensembles D_I et D_O ne sont pas nécessairement finis. Cependant, les modèles que nous construirons par apprentissage resteront des approximations finies, dans lesquels les valeurs explorées resteront dans des sous-domaines finis, avec dans tous les cas un ensemble fini de transitions.

Nous imposons néanmoins les restrictions suivantes aux machines que nous considérerons : nous supposons qu'elles seront complètes, déterministes, et observables.

Propriété 1 (Complet) *Un automate PFSM est complet si $\forall q \in Q, \forall i \in I$ et $\forall x \in D_I, \exists t \in T$ tel que $t = (q, q', i, o, p, f)$, avec $x \in p$.*

Comme pour les machines de Mealy simples, on peut compléter un automate paramétré en ajoutant des transitions bouclant sur l'état pour toutes les entrées non acceptées dans cet état. Ces transitions contiennent un symbole de sortie spécial Ω .

Propriété 2 (Déterministe) *Un automate paramétré est déterministe pour les entrées si pour $t_1, t_2 \in T$ tels que $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$ et $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ et $t_1 \neq t_2$, si $q_1 = q_2 \wedge i_1 = i_2$ alors $p_1 \cap p_2 = \emptyset$.*

En d'autres termes, un automate est déterministe si pour une entrée donnée et une valeur du paramètre donnée, une seule transition lui est applicable dans un état donné.

Propriété 3 (Observable) *Un automate paramétré est observable si pour $t_1, t_2 \in T$ tels que $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$ et $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ et $t_1 \neq t_2$, alors si $q_1 = q_2 \wedge i_1 = i_2$ alors $o_1 \neq o_2$.*

Dans un automate observable, deux transitions différentes partant du même état pour le même symbole d'entrée conduisent à deux symboles de sorties différents, ce qui permet de les différencier et de savoir que l'automate a pris un chemin différent. On peut aussi parler de non-déterminisme (au sens où il y a deux transitions) observable.

Quand M est dans l'état $q \in Q$ et reçoit l'entrée $i \in I$ portant la valeur de paramètre $x \in D_I$, alors l'état d'arrivée q' , le symbole de sortie o et la valeur associée par f sont déterminés par les fonctions δ , λ and σ respectivement, définies comme suite :

- $\delta : Q \times I \times D_I \longrightarrow Q$ est la fonction d'état d'arrivée; i.e. $\delta(q, i, x) = q'$ t.q. $(q, q', i, o, p, f) \in T$ et $x \in p$
- $\lambda : Q \times I \times D_I \longrightarrow O$ est la fonction de sortie
- $\sigma : Q \times I \longrightarrow D_O^{D_I}$ est la fonction définissant le paramètre de sortie.

Pour une séquence d'entrée $\gamma = i_1, \dots, i_k$ et une séquence de valeurs des paramètres $\alpha = x_1, \dots, x_k$, où chaque $i_j \in I, x_j \in D_I, 1 \leq j \leq k$, nous définissons l'association de γ à α comme $\gamma \otimes \alpha = i_1(x_1), \dots, i_k(x_k)$, où chaque x_j est associé

à i_j . On procède de même pour associer les valeurs des paramètres de sortie aux symboles de sortie.

Notre modèle présente les extensions suivantes par rapport aux modèles d'automates habituellement pris en compte dans l'inférence de machines.

- Des entrées et sorties paramétrées.
- Des domaines arbitraires (non nécessairement finis) pour les paramètres.
- Des gardes p sur les paramètres d'entrée; ces gardes sont associées aux transitions.
- Des fonctions arbitraires f pour le calcul des valeurs des paramètres en sortie.
- Ces fonctions peuvent être partielles.

Il y a cependant quelques restrictions dans ce modèle si on le compare aux modèles d'automates étendus du genre EFSM, tels qu'on les trouve dans des formalismes comme Estelle, SDL ou les Statecharts.

- Un paramètre unique pour les entrées et les sorties, et un domaine unique commun à tous les symboles.
- Pas de variables (dite parfois variables d'états). Toute la mémorisation doit être codée dans les états de Q .

Le premier point n'est pas une restriction sévère, c'est une commodité de notation. En effet, comme nous permettons des domaines arbitraires, il suffit d'introduire un codage entre les produits des domaines typés de paramètres des interactions réelles avec le composant et un domaine unique de représentation.

La seconde restriction est plus gênante, mais vient de l'impossibilité d'observer la structure interne de la boîte noire. Si un automate avait à la fois un état et une ou des variables internes, un même comportement pourrait être décrit par un automate à un seul état dans lequel toute la structure de contrôle serait codée dans une manipulation des variables, ou qui pourrait avoir un nombre arbitraire d'états. Il n'y aurait donc pas unicité de la solution calculée, ce qui remettrait en cause toute la démarche algorithmique.

Cette seconde restriction est donc sérieuse, puisqu'elle ne permet d'inférer le modèle que pour des composants ayant des variables dans des domaines finis, et de faible taille, car elle conduit à énumérer les états. Cependant, grâce au modèle paramétré que nous proposons, une forte réduction de la taille des automates inférés est déjà acquise par rapport aux algorithmes existants qui n'infèrent que des automates accepteurs déterministes, puisque nous pouvons factoriser un grand nombre de transitions. Pour aller plus loin et factoriser des états, nous pensons qu'on pourrait recourir à des heuristiques.

Comme nous n'inférerons qu'une approximation des composants, nous ne prendrons en compte qu'un nombre fini d'états. Nous devons supposer par ailleurs que les composants étudiés ont un modèle PFSM.

Dans un travail appuyé sur le code source (à des fins de compréhension des programmes), [16] proposent une technique permettant de reconstituer des structures d'automates enrichis avec une mémoire structurée. Un modèle et un algorithme d'inférence d'automates proposant certaines formes de paramètres a été proposé par [2]. Mais ce modèle ne prend en compte que des paramètres

booléens (et donc des domaines finis), et ne comporte pas de sorties (c'est une extension des automates accepteurs traités par Angluin).

5 Apprentissage de systèmes paramétrés

Cette partie explique comment on peut reconstituer un modèle d'automate paramétré de type PFSM présenté en 4. On présente les principes de l'algorithme, sans rentrer dans les détails. Comme le modèle PFSM est assez général pour représenter un composant réactif ayant un nombre fini d'états, on considère qu'on cherche à apprendre (ou identifier) un automate de référence inconnu $M = (Q, I, O, D_I, D_O, T, q_0)$, dont on connaît l'alphabet d'entrée I et le domaine du paramètre d'entrée D_I . Puisque nous pouvons soumettre n'importe quelle séquence d'entrées paramétrées au composant et observer les sorties paramétrées correspondantes, alors pour toute séquence d'entrée $\gamma \otimes \alpha (\gamma \in I^*, \alpha \in D_I^*, |\gamma| = |\alpha|)$, $\lambda(q_0, \gamma, \alpha)$ peut être trouvé par le test. On suppose là encore que tout composant peut être réinitialisé avant chaque test.

5.1 Table d'observation étendue

Dans l'algorithme présenté en 3 pour les automates de Mealy, on a utilisé la structure de table d'observation pour enregistrer les sorties du composant. Pour les PFSM, il faut enregistrer non seulement les symboles de sortie, mais aussi les valeurs des paramètres. Nous notons cette table (S, E, R, TO) .

S est un ensemble fini non-vide de *séquences d'accès*, qui permettront effectivement d'accéder aux différents états du modèle. Dans un automate paramétré, l'état d'arrivée n'est pas déterminé seulement par la séquence de symboles d'entrée, mais aussi par la séquence des paramètres d'entrée. Nous appellerons l'association de ces deux séquences une *séquence d'entrée composée*; ce sont ces séquences composées qui constitueront nos séquences d'accès.

E est un ensemble fini non-vide et clos par suffixe de séquences de symboles d'entrée. Ce sont les *séquences de séparation*, car elles servent à discriminer les états de la conjecture.

R est un sur-ensemble de S . Chaque fois qu'on ajoute une séquence d'accès à S , on obtient un groupe de séquences d'entrées composées en étendant la séquence d'entrée avec tous les $i \in I$ et en choisissant des $x \in D_I$ étendant l'ensemble des valeurs des paramètres des séquences d'entrée; les séquences d'entrées composées sont ajoutées à R .

La fonction TO est définie sur $R \times E$. Dans le cas des PFSM, partant d'une même séquence d'entrée, on peut observer différentes séquences de sortie selon les valeurs qu'on donne aux paramètres d'entrée. Pour $r \in R$ et $e \in E$, la case $TO(r, e)$ contient l'association entre les paramètres des séquences d'entrée et ceux des séquences de sortie. La table 2 est un exemple d'une telle table d'observation, où la première colonne contient les séquences de R, avec S dans la première partie de la colonne (séparée par la barre horizontale).

5.2 Propriétés des tables d'observation

Dans le cas des automates simples de Mealy, on pouvait construire une conjecture dès que la table d'observation était *close* et *cohérente*. Pour les PFSM, la table doit avoir des propriétés supplémentaires pour construire une machine paramétrée en accord avec les observations des valeurs. Ces propriétés doivent permettre de comparer les lignes des tables, afin que la conjecture puisse être bien définie et minimale.

Les séquences de S représentent des états potentiels. Pour $r_1, r_2 \in R$, si l'on obtient, en partant des états atteints par r_1 et r_2 , des séquences de sortie paramétrées différentes lorsqu'on a fourni en entrée les mêmes séquences composées, alors les lignes r_1 et r_2 sont *dissemblables*. Dans ce cas, on sait que r_1 et r_2 correspondent à des états différents.

Si deux lignes ne sont pas dissemblables, il faut, pour pouvoir les comparer, avoir exécuté les mêmes groupes de séquences composées à partir des états correspondants. Une table d'observation dont toutes les paires de lignes satisfont cette propriété sera dite *équilibrée*.

Pour tout $s \in S$ et $e \in E$, si $TO(s \cdot e)$ contient plus d'un symbole de sortie, alors la ligne s sera dite *contestée*. Une ligne s peut être contestée si $s \cdot e$ a été testé avec des valeurs différentes pour les paramètres d'entrée. Dans ce cas, on ajoutera de nouvelles lignes à R , qu'on testera avec les valeurs de paramètres qui sont dans $TO(s \cdot e)$.

Lorsqu'une table est équilibrée, on peut utiliser la relation d'égalité simple "=" pour comparer les lignes. On peut alors utiliser les concepts de table *close* et *cohérente*.

Quand la table d'observation est équilibrée, *close* et *cohérente*, on peut proposer une PFSM de conjecture d'une façon analogue à celle utilisée pour les automates de Mealy, en prenant soin d'associer les correspondances entre valeurs des paramètres en entrée et en sortie.

5.3 Algorithme d'apprentissage pour PFSM

L'algorithme d'apprentissage d'un modèle PFSM est défini par les étapes suivantes.

1. Au départ $R = S = \emptyset$ et $E = I$. Les cellules de la table sont toutes remplies avec l'ensemble vide.
2. Ajouter $\epsilon \otimes \epsilon$ à S , ainsi que dans R .
3. Construire des cas de test paramétrés pour les cellules non encore remplies de la table, et exécuter ces séquences. Pour $r \in R$ et $e \in E$, le cas de test correspondant a pour préfixe r , pour suffixe e et comme séquence de paramètres d'entrée α de D_I^+ et le test $r \cdot e \otimes \alpha$.³
4. Enregistrer le résultat du test $r \cdot e \otimes \alpha$ dans la table. Seule la dernière partie de la séquence de sortie sera enregistrée, celle qui correspond à la longueur

³ On choisit la séquence de valeurs α parmi les valeurs extraites des scénarios d'usage.

de e (comme dans le cas des machines de Mealy). Le résultat du cas de test $\eta \otimes \beta$, où $\eta \in O^+$ est la séquence de sortie et $\beta \in D_O^+$ est la séquence de paramètres de sortie correspondante, est enregistré comme $(\alpha', \eta' \otimes \beta')$, où α' est la partie finale de α , η' celle de η , β' celle de β et $|\alpha'| = |\eta'| = |\beta'|$.

5. Équilibrer la table. Lorsqu'elle ne l'est pas, c'est à dire lorsque pour certaines séquences d'entrée les comportements du système pour différentes valeurs des paramètres d'entrée ne sont pas connus, construire les cas de test correspondants, les exécuter et enregistrer les résultats dans la table.
6. Pour chaque cas de test $r \cdot e \otimes \alpha$, s'il existe $r' \in R, e' \in E$ tel que $r' \cdot e'$ constitue un préfixe de $r \cdot e$, alors le préfixe correspondant des valeurs des paramètres sera rajouté à $TO(r' \cdot e')$. Si r' devient une ligne contestée, alors on ajoute de nouvelles lignes dans R , que l'on teste avec les valeurs de paramètres enregistrées dans $TO(r' \cdot e')$.
7. Fermer la table. Chaque fois qu'elle n'est pas *close*, ajouter la séquence composée correspondante à S et revenir à l'étape 3 pour remplir les cases ajoutées.
8. Rendre la table *cohérente*. Lorsqu'elle ne l'est pas, ajouter la séquence d'entrée correspondante à E et revenir à l'étape 3 pour remplir les cases vides.
9. Lorsque la table est *équilibrée*, *close* et *cohérente*, on peut établir la conjecture M' .

5.4 Apprentissage du contrôleur de climatiseur comme système paramétré

Pour identifier le modèle PFSM du contrôleur de climatiseur, on peut utiliser les entrées indiquées sur la figure 1, sans adaptation particulière, comme dans le cas de l'apprentissage du modèle de Mealy. Ainsi, nous construisons $I = \{M, A, T\}$ avec $D_I = [-20, 50]$ comme domaine pour le paramètre d'entrée. On utilise l'algorithme présenté en 5. La table 2 est *équilibrée*, *close* et *cohérente*. La conjecture est illustrée sur la figure 4 avec les valeurs des paramètres déterminées par l'apprentissage.

Par exemple, les valeurs 4, 12, 20 et 35 correspondraient à des valeurs dans des plages dont la spécification (sous forme de scénarios d'usage) nous apprendrait qu'elles pourraient donner lieu à des comportements distincts. Noter que ces valeurs ne correspondent pas forcément aux valeurs limites de ces plages, et que sauf dans l'état 2 où les 4 valeurs donnent lieu à 4 transitions différentes, pour les autres états, seule une des (plages de) valeur introduit un comportement spécifique.

On voit bien qu'on a appris plus de détails que sur le modèle de Mealy. Grâce à la structure paramétrée de PFSM, nous sommes capables de tester des valeurs différentes pendant l'algorithme d'inférence. Nous venons de comprendre que quand la valeur de la température est 35°C, le contrôleur envoie une commande pour lancer la climatisation CM à moyenne vitesse mv . Par ailleurs, la vitesse de chauffage C est régulée quand la valeur de la température est 4°C. En revanche,

si la température est 20°C, il envoie hv pour mettre en route le ventilateur V à vitesse haute. La conjecture ne recouvre pas tout le domaine du paramètre, mais ne connaît que les valeurs qui ont été effectivement testées. En effet, on ne peut parcourir dans le test tout le domaine des paramètres. Même s’il est fini, la combinaison des valeurs demanderait trop de cas de test, on ferait du test exhaustif.

R	M	A	T
$\epsilon \otimes \epsilon$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M \otimes \perp$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T \otimes \perp - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, F \otimes \perp), (35, F \otimes \perp)$
$M - T \otimes \perp - 20$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, V \otimes hv), (35, F \otimes \perp)$
$M - T \otimes \perp - 35$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, F \otimes \perp), (35, CM \otimes mv)$
$M - T \otimes \perp - 12$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - A \otimes \perp - 4 - \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M - T - A \otimes \perp - 20 - \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M - T - A \otimes \perp - 35 - \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M - T - T \otimes \perp - 4 - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, F \otimes \perp), (35, F \otimes \perp)$
$M - T - T \otimes \perp - 4 - 20$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \perp - 20 - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \perp - 20 - 20$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, V \otimes hv), (35, F \otimes \perp)$
$M - T - T \otimes \perp - 35 - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \perp - 35 - 35$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, F \otimes \perp), (35, CM \otimes mv)$

TAB. 2. Une table *close* et *cohérente* pour apprendre un modèle PFSM du contrôleur de climatiseur

6 Test d’intégration

Après avoir présenté l’inférence de modèles pour les composants, nous décrivons ici comment exploiter ces algorithmes et ces modèles dans le développement d’un système par intégration de composants. On suppose que l’intégrateur logiciel se situe dans le cadre présenté en introduction, avec les hypothèses que nous y avons faites.

Dans une phase préparatoire de test “unitaire” des composants isolés, on construit pour chaque composant C un premier modèle PFSM $C^{(1)}$ selon l’algorithme décrit en 5. Ensuite, on assemble les composants d’un côté et leurs modèles de l’autre : les sorties d’un composant peuvent devenir les entrées d’un autre.

L’intégration se fait en deux étapes. Dans une première étape, on fournit au système (assemblé) les scénarios d’usage du système global (au besoin, si on ne dispose pas de tels scénarios, on peut en engendrer à partir de la définition d’interfaces). Les scénarios devront comporter normalement une instanciation des séquences d’entrées et de sortie avec des valeurs typiques des paramètres, ou des domaines de valeurs dans lesquelles on pourra choisir des valeurs pour les entrées. On fournit au système et à son modèle les séquences d’entrées paramétrées et on observe les sorties paramétrées. A ce stade, si les sorties ne sont pas celles

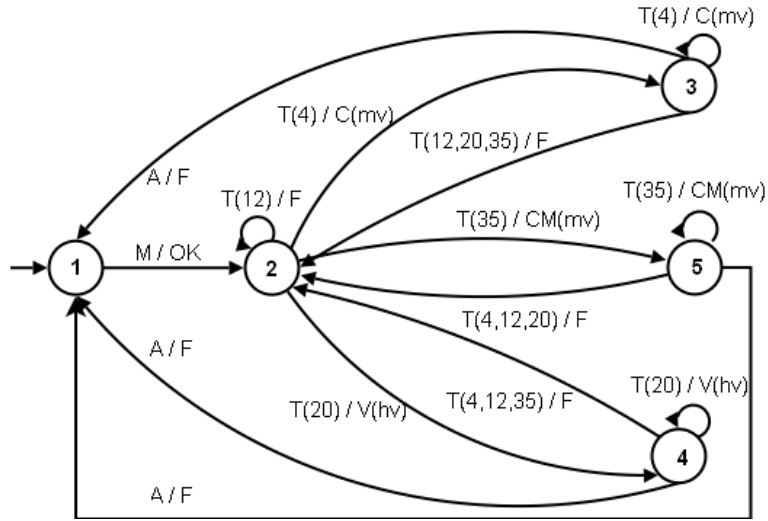


FIG. 4. Conjecture de PFMSM

attendues (lorsqu'elles ont été spécifiées), une erreur est détectée, ou si elles diffèrent de celles produites par le modèle, on est en présence d'un contre-exemple qui sera réinjecté dans l'apprentissage pour enrichir les modèles des composants. Une première phase d'enrichissement incrémental des modèles des composants peut donc avoir lieu au cours de cette première étape.

Dans une deuxième étape, on procède au test d'interopérabilité, pour lequel on s'appuie sur les modèles, comme générateurs. Pour cela, on utilise des techniques de génération par exploration de modèles comme celles de [14], [3], [10] etc. Le choix des paramètres associés aux entrées externes du système sera fait en tenant compte des valeurs typiques fournies avec les scénarios d'usage, en essayant d'enrichir les cas déjà testés dans la première étape. Les sorties paramétrées (tant internes qu'externes) observées en testant le système sont confrontées à celles prévues par les modèles. Le test continue jusqu'à ce qu'une sortie soit incompatible avec le modèle (symbole de sortie ou paramètre différent de celui prévu par le modèle) ou qu'un critère de couverture défini pour l'arrêt du test soit atteint. Si une incompatibilité a été détectée, elle fournit un contre-exemple pour raffiner le modèle. L'expert procédant à l'intégration peut également être sollicité pour décider si le contre-exemple est un comportement admissible ou erroné du système intégré.

Les contre-exemples obtenus dans les deux étapes peuvent servir à raffiner les modèles. Ils sont alors injectés dans l'algorithme d'apprentissage comme séquences rajoutées à S pour les composants concernés. Les valeurs des paramètres compatibles avec les modèles sont également prises en compte à ce niveau pour enrichir les tables. On produit alors une nouvelle version du modèle du système avec les des modèles $C^{(i+1)}$ pour chaque composant C .

À la fin du processus d'intégration, on dispose donc d'un modèle paramétré pour chacun des composants qui est compatible avec tous les tests exécutés. Les interactions entre tous les composants auront été systématiquement testés selon le critère de sélection de test et de couverture que l'intégrateur se sera donné. Le processus de test aura également pu être interrompu en cas de découverte de fautes dans le système. Pour plus de détails sur l'algorithme d'intégration, voir [11]

7 Conclusion

Nous avons présenté une approche destinée à assister l'intégrateur de composants logiciels. Prenant acte de l'absence habituelle de modèles formels accompagnant les composants, nous proposons une méthode et des algorithmes permettant d'utiliser les techniques de génération de tests à partir de modèles, en s'appuyant sur des modèles eux-mêmes reconstruits et enrichis au cours du test. Comme les modèles sont eux-mêmes déduits des observations faites au cours des tests, ils ne peuvent servir d'oracles absolus. Mais ils permettent de dériver et de tester systématiquement les interactions entre les composants.

La méthode s'inscrit bien dans une assistance à base formelle au développement logiciel. L'ingénieur dispose d'outils automatiques pour construire les modèles, dériver et exécuter les tests. Il reste maître de l'architecture du système, de la définition des scénarios d'usage, et de l'analyse des fautes et des divergences entre le modèle et le système réel.

Comme les difficultés d'intégration entre composants provenant de sources différentes sont souvent liées à des discordances dans le traitement des valeurs échangées entre les composants, nous accordons un intérêt particulier à la construction de modèles paramétrés. Pour cela, nous avons développé de nouveaux algorithmes d'inférence de machines pour des modèles de ce type.

Nous poursuivons ce travail dans plusieurs directions. D'abord, bien qu'il soit plus expressif que les automates finis étudiés jusqu'ici en inférence de machines (à l'exception de [2]), notre modèle reste limité par l'absence de variables internes, ce qui ne permet de reconnaître que des machines ayant un nombre fini d'états. On pourrait envisager des modèles plus proches d'EFSM, éventuellement en supposant que l'intégrateur est capable de fournir des informations sur la structure de la machine. Le problème d'inférence de telles machines sera cependant difficile. Nous avons développé un outil, RALT, qui met en oeuvre les algorithmes d'inférence pour les automates accepteurs, les machines de Mealy et les automates paramétrés. Nous devrions l'appliquer prochainement à des cas d'étude issus des services de télécommunication développés par France Télécom. Au-delà du travail sur les modèles et l'approche de test incrémental, nous sommes également intéressés à quantifier, ou du moins à qualifier plus précisément, le niveau de confiance à accorder à l'assemblage à l'issue (et au cours) du processus d'intégration. Pour cela, nous travaillons sur la définition d'une notion d'approximation compatible avec notre approche. Enfin, la démarche de test est

paramétrée par les critères de couverture, et nous souhaitons développer la mise en relation formelle de ces critères avec l'approximation réalisée par les modèles.

Références

1. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2 :87–106, 1987.
2. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
3. C. Besse, A. Cavalli, M. Kim, and F. Zaidi. Two methods for interoperability tests generation : An application to the tcp/ip protocol. In *Proceedings of TestCom 2002*, Berlin, 2002.
4. Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *FORTE'06*, 2006.
5. Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, 2002.
6. Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Fundamental Approaches to Software Engineering*, pages 80–95, 2002.
7. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.
8. O. Koné and R. Castanet. Test Generation for Interworking Systems. *Computer Communications*, 23(7) :642–652, 2000.
9. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.
10. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2006.
11. Erkki Makinen and Tarja Systa. MAS - an interactive synthesizer to support behavioral modelling in UML. In *ICSE '01 : Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.
12. Clémentine Nebut, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test generation : A use case-driven approach. *IEEE Trans. Softw. Eng.*, 32(3) :140, 2006.
13. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proceedings of FORTE'99*, Beijing, China, 1999.
14. A. Petrenko and N. Yevtushenko. Solving asynchronous equations. In *FORTE'98*, France, 1998.
15. Stephane S. Somé. Beyond scenarios : generating state models from use cases. In *Proceedings of SCESM*, 2002.
16. Neil Walkinshaw, Kirill Bogdanov, and Mike Holcombe. Identifying state transitions and their functions in source code. In *TAIC PART*, pages 49–58. IEEE Computer Society, 2006.