Adaptation des Protocoles des Composants par les Automates d'Interface

Samir Chouali, Sebti Mouelhi, Hassan Mountassir
Laboratoire d'Informatique de l'université de Franche-Comté, LIFC
{schouali, smouelhi, hmountassir }@lifc.univ-fcomte.fr

Résumé

Un des objectifs de l'ingénierie des logiciels à base de composants (CBSE) est de permettre la réutilisation des composants sans affecter leurs implémantations. Pour atteindre cet objectif, il est nécessaire de proposer des méthodes et des outils d'adaptation des composants avec leur environnement lorsque des incompatibilités se produisent au cours de leurs interactions. Dans ce papier, nous proposons une approche formelle d'adaptation des composants dont les protocoles comportementaux sont décrits par les automates d'interface, afin d'éliminer les disparités entre les composants aux niveaux des signatures et des protocoles. L'approche proposée tire profit de l'approche optimiste des automates d'interface. Ce formalisme permet de spécifier l'ordonnancement temporel des services requis et offerts des composants.

Mots clés : modèle à composants, protocoles, adaptation, réutilisation.

1 Introduction

Les systèmes à base de composants sont constitués d'un ensemble d'entités communicantes, appelées *composants*. L'idée de l'ingénierie des logiciels à base de composants [17, 8] est de développer des applications logicielles par l'assemblage de divers composants. Cette approche du développement permet la réutilisation de logiciels sans modifier le code des composants. Par conséquent, on économise le coût et le temps de développement.

Un composant est une unité logicielle dont les interfaces sont spécifiées par des contrats et des dépendances explicites à son environnement [17]. Une interface décrit les services offerts et requis par un composant. Elle représente les seules informations visibles du composant. Les interfaces donnent des informations sur le composant aux niveaux de la signature de ses méthodes, de son protocole comportemental, de la sémantique de ses opérations, et de la qualité de ses services. L'interopérabilité peut être définie comme l'aptitude à faire communiquer et coopérer deux ou plusieurs composants indépendamment de leurs codes, leur environnement d'exécution, ou leurs modèles d'abstraction [10, 19].

Habituellement, l'interopérabilité des composants n'est pas garantie lors de leur assemblage. Cela est dû à l'incompatibilité possible qui peut se produire entre les composants à différents niveaux [6]: leurs signatures, leurs comportements, la sémantique

de leurs opérations et la qualité de service. Lorsque deux composants sont incompatibles, des composants adaptateurs doivent être générés afin d'éliminer les anomalies qui peuvent se produire. Les adaptateurs sont des composants capables de résoudre le problème d'incompatibilité entre les composants dont les interfaces ne sont pas compatibles. Donc, nous considérons les adaptateurs comme étant des composants qui doivent être générés automatiquement pour assurer un assemblage valide des composants. Les plateformes orientées composants, comme CORBA et Fractal ou .NET, etc. peuvent détecter la non-correspondance entre les composants seulement au niveau des signatures de leurs actions (noms des méthodes et messages). L'adaptation au niveau du protocole et de la sémantique des opérations reste insatisfaite.

Dans ce papier, nous mettons l'accent sur l'adaptation des composants dont les interfaces sont décrites par les *automates d'interface* [1, 2, 3]. Le formalisme a été introduit par L. Alfaro et T. Henzinger. Il permet de décrire un ordre temporel sur l'enchaînement des appels des services requis et offerts. Les opérations locales d'un composant sont appelées actions internes. La composition de deux automates d'interface est réalisée en synchronisant leurs actions partagées d'entrée/sortie. Cette approche permet aussi de détecter les incompatibilités entre deux automates d'interface. L'incompatibilité est détectée lorsque, l'enchaînement des actions dans le produit synchronisé des deux automates conduit vers des états illégaux. Ces états signifient que l'un des deux composants sollicite un service qui n'est pas offert par l'autre composant. Quand deux composants sont incompatibles, ils le sont dans tous les environnements possibles.

Notre objectif est de générer automatiquement un adaptateur pour deux automates d'interface selon un *mapping* qui est un ensemble de règles sur les actions non partagées des deux composants. En ce sens, ce sont tout simplement des vecteurs de synchronisation. L'adaptateur devient un composant médiateur qui permet de résoudre les incompatibilités de comportements entre deux composants. L'adaptateur est considéré comme un automate d'interface au milieu permettant d'éliminer des anomalies aux niveaux des signature et des protocoles de deux composant.

L'une des contributions de ce papier est d'utiliser la notion d'adaptation des automates d'interface et leur approche optimiste de composition. En effet, deux automates d'interface A_1 et A_2 sont compatibles s'il existe un environnement qui les empêchent d'atteindre des états illégaux de leur produit synchronisé. Alors que dans les approches classiques (considérées pessimistes) A_1 et A_2 seraient incompatibles.

Ce papier est organisé de la manière suivante. Dans la section 2, nous présentons le modèle des automates d'interface. Dans la section 3, nous décrivons l'incompatibilité entre les composants dont les contrats sont décrits par des protocoles de comportement. Dans la section 4, nous présentons notre spécification de l'adaptation des composants en utilisant les automates d'interface. Dans la section 5, nous présentons l'algorithme de la construction de l'adaptateur qui est généré automatiquement. Les travaux connexes à notre approche, la conclusion, et les perspectives sont présentés dans les sections 6 et 7.

2 Automates d'interface

Les automates d'interface ont été introduits par Alfaro et Henzinger [1] pour modéliser les interfaces des composants. Ces automates sont issus des automates Input/Output [11] où il n'est pas nécessaire d'avoir des actions d'entrée activables dans tous les états. Chaque composant est décrit par un seul automate d'interface. L'ensemble des actions est décomposé en trois ensembles : les actions d'entrée, les actions de sortie, et les actions internes. Les actions d'entrée permettent la modélisation des méthodes qui vont être appelées dans un composant, dans ce cas elles représentent les services offerts pour un composant. Elles peuvent aussi modéliser une réception de messages dans un canal de communication. Ces actions sont étiquetées par le caractère "?". Les actions de sortie modélisent les appels des méthodes d'un autre composant. Donc, elles représentent les services requis par un composant. Elles peuvent aussi modéliser la transmission de messages dans un canal de communication. Ces actions sont étiquetées par le carcatère "!". Les actions internes sont des opérations activables localement et elles sont étiquetées par le caractère ";".

Définition 1 (Automate d'Interface). *Un* automate d'interface A *est représenté par le tuple* $\langle\ S_A,\ I_A,\ \Sigma_A^I,\ \Sigma_A^O,\ \Sigma_A^H,\ \delta_A\ \rangle$ *tels que :*

- $_A$ *est ensemble fini d'états*;
- $I_A \subseteq S_A$ *est un sous ensemble des états initiaux de cardinalité* $card(I_A) \leq 1$;
- Σ_A^I, Σ_A^O *and* Σ_A^H*, resprésentent, respectivement, les ensembles des actions d'entrée, de sortie et internes. L'ensemble des actions de* A *est noté par* Σ_A;
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *est l'ensemble des transitions entre les états.*

Les actions d'entrée et de sortie d'un automate d'interface A sont appelées actions externes ($\Sigma_A^{ext} = \Sigma_A^I \cup \Sigma_A^O$). Les actions de sortie et internes sont appelées actions localement controlées ($\Sigma_A^{loc} = \Sigma_A^O \cup \Sigma_A^H$). L'ensembles des actions internes Σ_A^H peut contenir l'action, *epsilon* ϵ, qui symbolise un événement non opérationnel. Nous définissons par $\Sigma_A^I(s)$, $\Sigma_A^O(s)$, $\Sigma_A^H(s)$, $\Sigma_A^{ext}(s)$ et $\Sigma_A^{loc}(s)$ respectivement les ensembles des actions d'entrée, de sortie, externes et internes activables à l'état s. $\Sigma_A(s)$ représente l'ensemble des actions activables de l'état s. Nous représentons par $\Sigma_A^*(s,a)$, l'état t tels que $(s,a,t) \in \delta_A$ et $* \in \{O,I,H,loc,ext\}$.

La vérification de l'assemblage de deux composants s'obtient en vérifiant la compatibité de leurs automates d'interface. Pour vérifier l'assemblage de deux composants C_1 et C_2, on vérifie s'il existe un environnement pour lequel il est possible d'assembler correctement C_1 et C_2. Cela se traduit par la *composition* de leurs automates d'interface et la vérification si cette dernière n'est pas vide.

Deux automates d'interface A_1 et A_2 sont *composables* si $\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \setminus \{\epsilon\} \cap \Sigma_{A_2} = \Sigma_{A_2}^H \setminus \{\epsilon\} \cap \Sigma_{A_1} = \emptyset$. $Shared(A_1,A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$ est

l'ensemble des actions partagées entre A_1 et A_2.

Définition 2 (Produit Synchronisé). *Soient A_1 et A_2 deux automates d'interface composables. Le* produit synchronisé *$A_1 \otimes A_2$ de A_1 et A_2 est défini par :*

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ et $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup Shared(A_1, A_2)$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ si
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

Deux automates d'interface pourraient être incompatibles à cause de l'existence des états illégaux dans leur produit synchronisé. Les états illégaux sont des états à partir lesquels une action de sortie partagée d'un automate ne peut pas être synchronisée avec la même action activée en entrée dans l'autre composant.

Définition 3 (Etats Illégaux). *Soient deux automates d'interface composables A_1 et A_2, l'ensemble des* états illégaux *$Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ est défini par $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in Shared(A_1, A_2)$. telle que la condition C est satisfaite$\}$*

$$C = \begin{pmatrix} a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2) \\ \vee \\ a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1) \end{pmatrix}$$

L'approche des automates d'interface est considérée comme une approche optimiste, car l'atteignabilité des états dans $Illegal(A_1, A_2)$ ne garantit pas l'incompatibilité des A_1 et A_2. En effet, dans cette approche on vérifie l'existence d'un environnement qui fournit les services appropriés au produit $A_1 \otimes A_2$ afin d'éviter les états illégaux. Les états dans lesquels l'environnement peut éviter l'atteignabilité des états illégaux sont appelés états compatibles, et sont définis par l'ensemble $Comp(A_1,A_2)$. Cet ensemble est calculé dans $A_1 \otimes A_2$ en éliminant les états illégaux, les états inatteignables et les états qui conduisant vers des états illégaux en passant par des actions internes ou des actions de sortie.

Définition 4 (Composition). *La* composition *$A_1 \parallel A_2$ de deux automates compatibles A_1 et A_2 est définie par (i) $S_{A_1 \| A_2} = Comp(A_1,A_2)$, (ii) $I_{A_1 \| A_2} = I_{A_1 \otimes A_2} \cap Comp(A_1,A_2)$, (iii) $\Sigma^*_{A_1 \| A_2} = \Sigma^*_{A_1 \otimes A_2}$ où $* \in \{O, I, H\}$, et (iv) $\delta_{A_1 \| A_2} = \delta_{A_1 \otimes A_2} \cap$*

$Comp(A_1,A_2) \times \Sigma_{A_1 \| A_2} \times Comp(A_1,A_2)$.

Deux automates d'interface A_1 et A_2 sont compatibles si et seulement si leur composition $A_1 \parallel A_2$ contient au moins un état atteignable. La complexité de cette méthode est linéaire en fonction de la taille des deux automates d'interface.

3 L'incompatibilité entre composants

Plusieurs définitions ont été proposées dans le domaine de l'analyse de l'intéropérabilité des composants. Ces définitions sont essentiellement dues à la réutilisation des composants qui pose souvent des problèmes d'incompatibilité tels que: (i) noms des méthodes ou des messages échangés entre les composants ne correspondent pas (ii) l'ordre des messages ou des actions d'un protocole d'un composant ne recouvre pas celui d'un autre, (iii) une action dans un composant n'a pas d'équivalent dans l'autre, ou correspond à plusieurs autres actions.

Pour les automates d'interface, l'incompatibilité aux niveaux signature et protocole des composants ne peut pas être détectée en appliquant le produit synchronisé entre deux automates d'interface composables comme il a été défini dans la Définition 2. Par conséquence, toutes les actions non partagées ne synchronisent pas et elles sont associées de façon asynchrone dans le produit. Pour résoudre cette contrainte, nous présentons une spécification de l'adaptation pour désigner les exigences de composition et les relations entre les actions non partagées de deux automates d'interface.

Afin de résoudre l'incompatibilité entre les signatures des actions, il est nécessaire de définir des règles de correspondance qui lient les actions non partagées utilisées dans les différents composants pour mettre en œuvre certaines interactions. Dans notre approche, les règles peuvent être établies exactement entre deux composants et elles contiennent que des actions non partagées. La spécification d'adaptation deux automates d'interface A_1 et A_2 est un ensemble de règles appelé *mapping* qui fournit la spécification minimale d'un composant médiateur appelé adaptateur entre A_1 et A_2.

Définition 5 (Régles d'adaptation et Mapping). *Une* règle d'adaptation α *de deux automates d'interface composables* A_1 *et* A_2*, est un couple* $\langle L_1, L_2 \rangle \in (2^{\Sigma^O_{A_1}} \times 2^{\Sigma^I_{A_2}}) \cup (2^{\Sigma^I_{A_1}} \times 2^{\Sigma^O_{A_2}})$[1] *tel que* $(L_1 \cup L_2) \cap Shared(A_1, A_2) = \emptyset$ *et si* $\mid L_1 \mid > 1$ *(ou* $\mid L_2 \mid > 1$*) alors* $\mid L_2 \mid = 1$ *(ou* $\mid L_1 \mid = 1$*);*

Un mapping $\Phi(A_1, A_2)$ *de deux automates d'interface composables* A_1 *et* A_2 *et un ensemble de règles* α_i*, pour* $1 \leq i \leq \mid \Phi(A_1, A_2) \mid$[2].

Selon la Définition 5, une règle dans notre approche supporte des correspondences "une-pour-une" et "une-pour-plusieurs" entre les actions. L'adaptation de deux automates fait correspondre en général soit une action ou un ensemble d'actions d'un automate

[1] Soit un ensemble S, 2^S et l'ensemble des partie de S.

[2] $\mid E \mid$ est la cardinalité d'un ensemble E.

avec une seule action de l'autre. Considérons deux composants *Client* et *Serveur* impliqués dans une procédure d'authentification. Supposons que le client s'authentifie en envoyant d'abord son nom d'utilisateur et ensuite un mot de passe alors que le serveur accepte les données en une action unique (c.f. Figure 1).

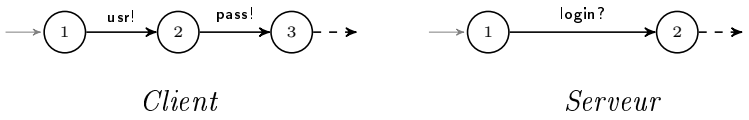


Figure 1: Une correspondence "une-pour-plusieurs" entre les actions

La notion d'adaptation permet de donner un sens à la synchronisation de certaines actions non partagés entre deux IAs composables A_1 et A_2 conformément au mapping $\Phi(A_1, A_2)$ donnée. Deux cas sont distingués : le premier cas, si $\Phi(A_1, A_2) = \emptyset$, alors A_1 et A_2 ne sont pas adaptables et leur synchronisation et définie par leur produit synchronisé classique $A_1 \otimes A_2$. Le deuxième cas, si $\Phi(A_1, A_2) \neq \emptyset$, leur synchronisation est réalisée en générant un automate Ad au milieu conformément à $\Phi(A_1, A_2)$. Nous noterons par $Mismatch_\Phi(A_1, A_2)$ l'ensemble $\{\mathrm{a} \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} \mid \exists\ \alpha \in \Phi(A_1, A_2)\ .\ \mathrm{a} \in \Pi_1(\alpha) \vee \mathrm{a} \in \Pi_2(\alpha)\}$[3].

Deux IAs A_1 et A_2 sont *adaptables* s'ils sont composables, leur mapping $\Phi(A_1, A_2)$ n'est pas vide et il existe un adaptateur Ad non vide assurant la communication entre eux. L'adaptateur est destiné à adapter les successions d'actions de sortie d'un automate présentes dans $\Phi(A_1, A_2)$ qui sont suivis par les actions d'entrée correspondantes dans l'autre. Ainsi, l'adaptateur ne peut adapter parfaitement que les exécutions des deux automates qui respectent cette propriété. Nous distinguons deux cas principaux :

- si l'adaptateur généré (en utilisant l'algorithme présenté dans la Section 5) conformément au mapping donnée est vide donc A_1 et A_2 ne peuvent pas être adaptés.
- si l'adaptateur généré est non vide alors
  - si Ad est compatible avec les A_1 et A_2 et $A_1 \parallel Ad \parallel A_2$ est non vide alors A_1 et A_2 peuvent interagir après leur adaptation.
  - sinon, A_1 et A_2 ne sont pas bien formés pour être adapté sans provoquer des incompatibilités (l'approche optimiste des IAs).

4 La spécification de l'adaptateur

Un adaptateur légal de deux IAs adaptables A_1 et A_2 conformément à un mapping $\Phi(A_1, A_2)$, est un IA Ad qui leur permet d'interopérer malgrè leur comportements incompatibles. L'adaptateur doit être composables avec A_1 et A_2. L'adaptateur doit

[3] $\Pi_1(\langle a, b\rangle) = a$ et $\Pi_2(\langle a, b\rangle) = b$ sont respectivement la projection sur le premier élément et le deuxième élément du couple $\langle a, b\rangle$.

être construit en respectant l'ordonnancement des événements de A_1 et A_2, en recevant toujours les actions de sortie d'un automate avant de délivrer des réponses à leurs correspondantes actions d'entrée de l'autre automate conformément à $\Phi(A_1, A_2)$.

Soit un automate d'interface A, nous noterons par $\Theta_A^S(s) \subseteq S_A^*$ l'ensemble des exécutions finies successeures $\theta = s_1 a_1 s_2 a_2 ... s_n$ telles que $s_1 = s$, s_n est l'état initial ou un état qui n'a pas de transitions sortante et pour tout $1 \leq i < n$, il y a une transition (s_i, a_i, s_{i+1}) $\in \delta_A$. Nous noterons par $\Theta_A^P(s) \subseteq S_A^*$ l'ensemble des exécutions finies prédecesseurs θ $= s_1 a_1 s_2 a_2 ... s_n$ est défini exactement comme $\Theta_A^S(s)$ sauf que $s_1 = i$ tel que $i \in I_A$ et $s_n = s$. L'ensemble Θ_A de toutes les exécutions de A égal à $\Theta_A^S(i)$ où $i \in I_A$. On dit qu'une succession de transitions $s_1 a_1 s_2 a_2 ... s_n$ $(n \geqslant 2)$ est *inclue* dans l'exécution σ dans $\Theta_A^S(s)$ ou $\Theta_A^P(s)$ (représentée pas l'opérateur $\sqsubseteq$), si toutes les transitions de $s_1 a_1 s_2 a_2 ... s_n$ sont des transitions de σ.

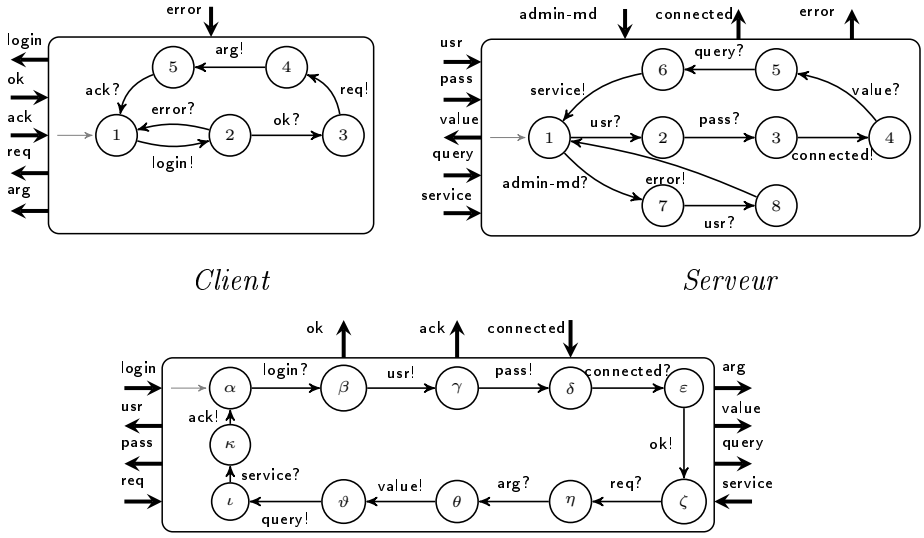


Figure 2: L'adaptateur Ad d'une variante d'un système client/serveur

Définition 6 (Adaptateur). *Soient deux automates d'interface adaptables A_1, A_2 et un mapping $\Phi(A_1, A_2)$ non vide. Un* adaptateur *de A_1 et A_2 conformément à $\Phi(A_1, A_2)$ est un automate d'interface $Ad = \langle\ S_{Ad},\ I_{Ad},\ \Sigma_{Ad}^I,\ \Sigma_{Ad}^O,\ \Sigma_{Ad}^H,\ \delta_{Ad}\ \rangle$ tel que*

- $\Sigma_{Ad}^I = \{a \in \Sigma_{A_1}^O \cup \Sigma_{A_2}^O \mid a \in \text{Mismatch}_\Phi(A_1, A_2)\}$;
- $\Sigma_{Ad}^O = \{a \in \Sigma_{A_1}^I \cup \Sigma_{A_2}^I \mid a \in \text{Mismatch}_\Phi(A_1, A_2)\}$;
- $\Sigma_{Ad}^H = \{\epsilon\}$;
- $\delta_{Ad} \subseteq S_{Ad} \times \Sigma_{Ad}^I \cup \Sigma_{Ad}^O \cup \{\epsilon\} \times S_{Ad}$;
- $Shared(Ad, A_1) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_1(\alpha)$;

- $Shared(Ad, A_2) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_2(\alpha)$.
- *Pour tout* $s \in S_{Ad}$ et $\sigma \in \Theta^P_{Ad}(s)$, *s'ils existent* $r_1 a_1 ... r_n a_n s \sqsubseteq \sigma$ *et* $\alpha \in \Phi(A_1, A_2)$ *telles que* $\Pi_i(\alpha) \subseteq \Sigma^O_{A_i}$ *pour* $i \in \{1, 2\}$ *et* $\Pi_i(\alpha) \subseteq \bigcup_{k \in 1..n} \{a_k\}$, *alors pour toute* $\rho \in \Theta^S_{Ad}(s)$, *il existe* $sb_1 ... b_m t_m \sqsubseteq \rho$ *telle que* $\Pi_{3-i}(\alpha) \subseteq \Sigma^I_{A_{3-i}}$ *et* $\Pi_{3-i}(\alpha) \subseteq \bigcup_{l \in 1..m} \{b_l\}$.

Propriété 1. Un adaptateur Ad de deux automates d'interface adaptables A_1 et A_2 conformément à un mapping $\Phi(A_1, A_2)$ est composable avec A_1 et A_2.

La propriété peut être vérifiée aisément selon les Définitions 2 et 6. La composition de A_1 et A_2 est réalisée en synchronisant d'abord Ad avec A_1 ou A_2, puis en calculant leur composition, et enfin en composant la composition résultante avec l'automate réstant. Si l'automate d'interface composite $A_1 \parallel Ad \parallel A_2$ est non vide alors A_1 et A_2 sont compatibles après l'adaptation.

Exemple 1. Soit les deux IAs *Client* et *Serveur* de la Figure 2 et le mapping $\Phi($ *Client*, *Serveur*$) = \{\langle\{login!\}, \{usr?, pass?\}\rangle, \langle\{req!\}, \{query?\}\rangle, \langle\{arg!\}, \{value?\}\rangle, \langle\{ack?\}, \{service!\}\rangle, \langle\{ok?\}, \{connected!\}\rangle\}$. L'action *admin-md*? est un signal reçu de l'administrateur du système pour ouvrir une session super-utilisateur. Quand un nom d'utilisateur d'un client est reçu par le serveur après avoir reçu le signal *admin-md*! de l'administrateur, une erreur est détectée.

L'adaptateur le plus convenable pour *Client* et *Serveur* est donné par l'adaptateur *Ad*. On peut conclure que *Ad* est composable et compatible à la fois avec *Client* et *Serveur* et il satisfait la Définition 6. L'adaptateur est conçu en fonction de l'ordonnancement des événements des deux interfaces *Client* et *Serveur* et il reçoit les actions de sortie de $\Phi($*Client*,*Serveur*$)$ avant d'émettre des actions de sortie pour leurs entrées correspondantes. Notre algorithme proposé dans la Section 5 génére le même automate *Ad* présenté dans la Figure 2.

L'état $2\gamma 8$ de $(Client \| Ad) \otimes Serveur$ est illégal. On appliquant l'alogrithme proposé in [1] sur le produit, nous déduisons que l'automate composite $(Client \| Ad) \| Serveur$ est non vide. Par conséquent, *Client* and *Serveur* deviennent compatibles à travers *Ad*.

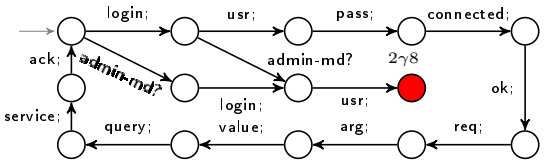


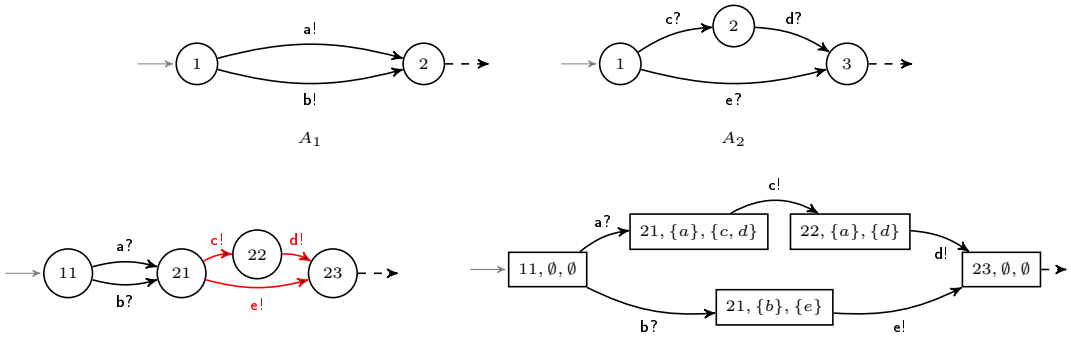
Figure 3: $(Client \parallel Ad) \otimes Server$

5 La génération de l'adaptateur

Dans cette section, nous présentons un algorithme permettant de construire un adaptateur pour deux IAs adaptables A_1, A_2 et un mapping $\Phi(A_1, A_2)$. L'algorithme présenté ci-dessous lit en parallèle A_1 et A_2 et construit au fur et à mesure l'ensemble des états et des transitions de l'adaptateur. L'algorithme est exécuté en respectant l'ordre des événements. Il ajoute les actions d'entrée de A_1 et A_2 qui sont présentent $\Phi(A_1, A_2)$ comme étant des actions de sortie, puis ajoute les actions de sortie correspondantes. L'algorithme doit construire un adaptateur en harmonie avec les deux interfaces A_1 et A_2 en respectant leurs spécifications.

L'algorithme

Algorithme 1 parcourt A_1 et A_2 en explorant alternativement leurs états et leurs transitions. Il met à jour au fur et à mesure l'ensemble S des états générés et l'ensemble T des transitions générées initialement égale à vide.



(a) mauvaise adaptation

(b) la solution

Figure 4: La forme des états de l'adaptateur

Un problème crucial que nous rencontrons c'est que nous ne pouvons pas être limités au produit des états $S_{A_1} \times S_{A_1}$ de A_1 et A_2 pour construire le système des transitions de leur adaptateur. Le problème se produit lorsque, à partir d'un état s_1 dans l'un des deux automates, deux actions de sortie a et b de $Mismatch\Phi(A_1, A_2)$ sont activées et elles atteignent le même état destination s'_1, et à partir d'un état s_2 dans l'autre, leurs actions d'entrée correspondantes sont activées sur des chemins indépendants. Si nous restreignons l'ensemble des états généré à $S_{A_1} \times S_{A_1}$, nous ne pouvons pas décider quelles actions nous devons activées à partir de du couple (s'_1, s_2), les actions d'entrée correspondantes à a ou b.

Pour résoudre ce problème, nous avons utilisé en complément du produit $S_{A_1} \times S_{A_2}$, l'ensemble des actions de sortie consommées et celles d'entrée restantes conformément au *mapping*. A titre d'exemple, considérons les deux comportements des deux IAs A_1 et A_2 de la Figure 4 où a, $b \in \Sigma^O_{A_1}$ et c, d, $e \in \Sigma^I_{A_2}$. Supposons que $\Phi(A_1, A_2) \supseteq$

$\{\langle\{a!\},\{c?,d?\}\rangle,\langle\{b!\},\{e?\}\rangle\}$, L'adaptation souhaitée de ces deux comportements est montré dans la Figure 4(b).

Pour parcourir en profondeur A_1 et A_2, on utilise une pile *stk* où les éléments sont des tuples $\langle s_1, s_2, mk_{A_1}, mk_{A_2}, m^O, m^I, atr\rangle$ tel que $s_1 \in S_{A_1}$ et $s_2 \in S_{A_2}$ sont les états courants lus par la boucle, mk_{A_1} et mk_{A_2} sont respectivement les mémoires des états marqués par l'itération, m^O et m^I sont respectivement les mémoires des actions d'entrée et de sortie lues de $Mismatch_\Phi(A_1,A_2)$ et *atr* est un indicateur indiquant lequel des deux automates sera parcouru au cours d'une itération A_1 ou A_2 ou les deux ensemble.

Dans un tuple de pile τ, soit une action qui appartient à $Mismatch_\Phi(A_1,A_2)$ ou non est activée à partir de $\tau.s_1$ ou $\tau.s_2$. Les transitions étiquetées par des actions $a \in (S_{A_1} \cup S_{A_1}) \setminus Mismatch_\Phi(A_1,A_2)$ sont remplacées par des transitions étiquetées par l'action interne ϵ; dans T. Une transition étiquetée par une action de sortie a dans $Mismatch_\Phi(A_1,A_2)$ correspond à une transition étiquetée par $a?$ dans T. Une transition étiquetée par une action d'entrée b dans $Mismatch_\Phi(A_1,A_2)$ correspond à une transition étiquetée par $b!$ si l'ensemble des actions de sortie correspondantes dans le mapping ont été traitées avant.

Les états de l'adaptateur sont représentés non seulement par $S_{A_1} \times S_{A_2}$, mais aussi par la mémoire des actions d'entrée et de sortie de $Mismatch_\Phi(A_1,A_2)$. Les éléments de S sont déduites des tuples de la pile. Pour un tuple de pile $\langle s_1, s_2, mk_{A_1}, mk_{A_2}, m^O, m^I, atr\rangle$, nous associons un tuple d'état $\langle t_1, t_2, n^O, n^I\rangle$ tel que $t_1 = s_1$, $t_2 = s_2$, $n^O = m^O$ et $n^I = m^I$. Nous disons que s_i est *marquée* si $s_i \in mk_{A_i}$ pour $i \in \{1,2\}$. On note par $état(\tau)$, le tuple d'état correspondant à un tuple de pile τ. Pour un tuple de pile τ, $état(\tau)$ est *bloquant* si $\tau.s_1$ et $\tau.s_2$ sont marqués et si leurs mémoires d'actions ne sont pas vides ($\tau.m^O \neq \emptyset \vee \tau.m^I \neq \emptyset$).

Algorithme 1 Genération_états_transitions
Entrées: *Deux IAs adaptables A_1, A_2, et un mapping $\Phi(A_1,A_2) \neq \emptyset$.*
Sorties: *Les deux ensembles S et T.*
Initialisation: *$T = S = \emptyset$ et une pile stk contenant le tuple $\langle i_1, i_2, \emptyset, \emptyset, \emptyset, \emptyset, \{A_1, A_2\}\rangle$ tel que $i_1 \in I_{A_1}$ et $i_2 \in I_{A_2}$.*

Tant que stk n'est pas vide faire :

1. $\tau \leftarrow tête(stk)$; $\nu \leftarrow état(\tau)$; $dépiler(stk)$;
2. ***si*** $(\tau.atr = \{A_1, A_2\}$ ***ou*** $\tau.atr = \{A_1\})$ ***et*** ν *n'est pas bloquant et* $\nu \notin S$ ***alors***
   (i) ***si*** $\nu.t_1$ *n'est pas marqué* ***alors pour chaque*** $(\tau.s_1, a, s_1') \in \delta_{A_1}$ ***faire***
   - ***si*** $\exists\alpha \in \Phi(A_1,A_2) \mid a \in \Pi_1(\alpha)$ ***alors***
     - ***si*** $a \in \Sigma^O_{A_1}$ ***alors*** Traitement_actions_sortie$(A_1, A_2, \alpha, a, stk, \tau, s_1')$; *ajouter la transition* $(\nu, a?, état(tête(stk)))$ *à* T;
     - ***si*** $a \in \Sigma^I_{A_1} \cap \tau.m^I$ ***alors*** Traitement_actions_entrée$(A_1, A_2, \alpha, a, stk, \tau, s_1')$; *ajouter la transition* $(\nu, a!, état(tête(stk)))$ *à* T;
   - ***sinon si*** $a \notin Mismatch_\Phi(A_1,A_2)$ ***alors*** *empiler*$(\langle s_1', \tau.s_2, \tau.mk_{A_1} \cup \{\tau.s_1\}, \tau.mk_{A_2}, \tau.m^O, \tau.m^I, \tau.atr\rangle, stk)$; *ajouter la transition* $(\nu, \epsilon;, état(tête(stk)))$ *à* T;

   (ii) ***si*** *stk reste unchangée malgrè que* $\Sigma_{A_1}(\tau.s_1) \neq \emptyset$ ***et*** $\tau.atr = \{A_1\}$ ***alors***
   - ***si*** τ *a été traité avant pour* A_2 $(atr = \{A_2\})$ ***alors*** *empiler*(τ', stk) *tel que* τ' *est comme* τ *à l'exception de* $\tau'.mk_{A_1} = \tau.mk_{A_1} \cup \{\tau.s_1\}$;
   - ***sinon*** *empiler*(τ', stk) *tel que* τ' *est comme* τ *à l'exception de* $\tau'.atr = \{A_2\}$;

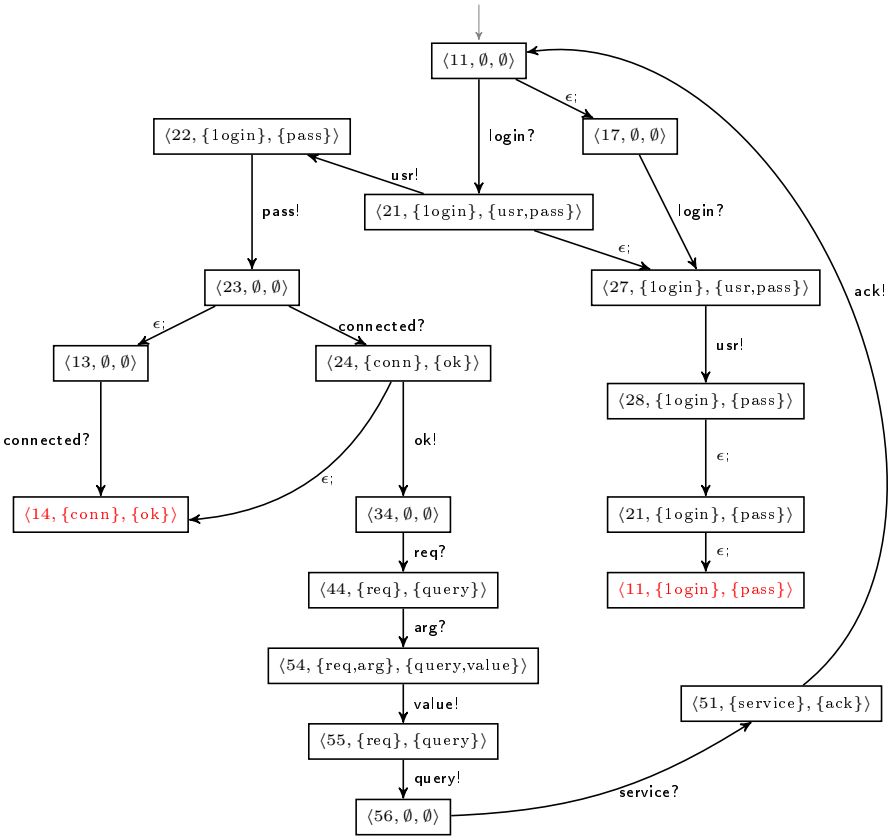


Figure 5: Le résultat de l'exécution de l'Algorithm 1 pour le système client/serveur de l'example 1

3. ***si*** $(\tau.atr = \{A_1, A_2\}$ ***ou*** $\tau.atr = \{A_2\})$ ***et*** $\nu.t_2$ *n'est pas bloquant* ***et*** $\nu \notin S$ ***alors*** *faire le même traitemet qu'avant en l'adaptant pour tout* $(\tau.s_2, b, s_2') \in \delta_{A_2}$;
4. ***si*** $\nu \notin S$ ***alors*** *ajouter* ν *à* S;

Fin Tant que

Les procédures du traitement des actions d'entrée et de sortie dans $Mismatch_\Phi(A_1,A_2)$ sont présentées ci-dessous.

Procédure Traitement_actions_sortie
Entrées: A_1, A_2, *une règle* ψ *de* $\Phi(A_1, A_2)$, $a \in Mismatch_\Phi(A_1, A_2)$, *stk, un tuple* τ *de stk et un état* t *de* $S_{A_1} \cup S_{A_2}$.

1. ***si*** $a \in \Sigma^O_{A_1}$ ***alors***
   - ***si*** $\tau.m^O \cup \{a\} \supseteq \Pi_1(\psi)$ ***alors*** *empiler*$(\langle t, \tau.s_2, \tau.mk_{A_1} \cup \{\tau.s_1\}, \tau.mk_{A_2}, \tau.m^O \cup \{a\}, \Pi_2(\psi) \cup \tau.m^I, \{A_2\}\rangle, stk)$;
   - ***sinon*** *empiler*$(\langle t, \tau.s_2, \tau.mk_{A_1} \cup \{\tau.s_1\}, \tau.mk_{A_2}, \tau.m^O \cup \{a\}, \tau.m^I, \{A_1\}\rangle, stk)$;

2. ***si*** $a \in \Sigma^O_{A_2}$ ***alors*** *faire le même traitement que 1 en l'adaptant pour* A_2;

Procedure Traitement_actions_entrée
Entrées: A_1, A_2, *une règle* ψ *de* $\Phi(A_1, A_2)$, $a \in Mismatch_\Phi(A_1, A_2)$, *stk, un tuple* τ *de stk et un état* t *de* $S_{A_1} \cup S_{A_2}$.

1. ***si*** $a \in \Sigma^I_{A_1}$ ***alors*** *soit* $m^O = \tau.m^O \setminus \Pi_2(\psi)$ *s'il n'y a pas d'actions réstantes de* $\Pi_1(\psi)$ *dans* $\tau.m^I \setminus \{a\}$. *sinon,* $m^O = \tau.m^O$;
   - ***si*** $\tau.m^I \setminus \{a\}$ *n'est pas vide* ***alors*** *empiler*($\langle$ t, $\tau.s_2$,$\tau.mk_{A_1} \cup \{\tau.s_1\}$,$\tau.mk_{A_2}$, m^O, $\tau.m^I \setminus \{a\}$, $\{B\}$ $\rangle$,*stk*) *tel que* B *est l'automate d'une action aléatoirement choisie de* $\tau.m^I \setminus \{a\}$;
   - ***si non*** *empiler*($\langle$ t, $\tau.s_2$,$\tau.mk_{A_1} \cup \{\tau.s_1\}$,$\tau.mk_{A_2}$, m^O, $\tau.m^I \setminus \{a\}$, $\{A_1, A_2\}$ $\rangle$,*stk*);
2. ***si*** $a \in \Sigma^I_{A_2}$ ***alors*** *faire le même traitement que 1 en l'adaptant pour* A_2;

Pour satisfaire la dernière condition de la Définition 6, les fragments des traces menant à des états bloquants doivent être retirés des transitions générées. L'Algorithme 2 supprime tous ces traces en parcourant le graphe résultant de l'Algorithme 1 en utilisant un parcour en arrière à partir des états bloquants jusuq'à les état qui ont plus q'une transition sortant. S'il ne reste aucun état, donc l'adaptateur de A_1 et A_2 n'est pas défini. Nous introduisons l'ensemble $Pre^1_T(B)$ $(B \subseteq S)$ des tuples d'états prédécesseurs μ dans S tel que $deg^o_T(\mu) = 1$[4] dans l'ensemble des transitions T.

Algorithme 2 Constructeur_adaptateur
Entrées: *L'ensemble des états* S *et l'ensemble des transitions* T *résultant de l'application de l'algorithme 1 dont les entrées sont deux IAs d'interface* A_1, A_2 *et un mapping* $\Phi(A_1, A_2)$.
Sorties: *L'adaptateur Ad de* A_1 *et* A_2 *conformément à* $\Phi(A_1, A_2)$ *tel que*
- $S_{Ad} = \{\langle \nu.t_1, \nu.t_1, \nu.n^O, \nu.n^I \rangle$ *tel que* $\nu \in S \setminus B_k\}$ *et*
- $\delta_{Ad} = \{(\langle \nu.t_1, \nu.t_1, \nu.n^O, \nu.n^I \rangle, a, \langle \nu'.t_1, \nu'.t_1, \nu'.n^O, \nu'.n^I \rangle) \in S_{Ad} \times \Sigma_{Ad} \times S_{Ad}$ *tel que*
ν, $\nu' \in T \setminus \{ (\mu, b, \mu') \in T \mid \mu, \mu' \in S \setminus B_k \} \}$.

Initialisation: $B_0 = \{\nu \in S$. ν *est bloquant*$\}$.

1. ***Pour*** $k \geqslant 0$, *let* $B_{k+1} = B_k \cup Pre^1_T(B_k)$;
2. *aller à l'étape* 1 ***tant que*** $B_k \subseteq B_{k+1}$;

En appliquant l'Algorithme 2 sur l'automate produit présenté dans la figure 5, nous obtenons l'adaptateur de la figure 2. L'algorithme effectue un parcours en arrière à partir des états illégaux $\langle 11, \{1ogin!\}, \{pass?\}\rangle$ et $\langle 14, \{connection!\}, \{ok?\}\rangle$ en éliminant les transitions jusqu'au états ayant plus d'une transition de sortie.

Pour deux automates d'interface A_1 et A_2, La complexité de l'algorithme 1 est de l'ordre de $\mathbf{O}(|S_{A_1} \times S_{A_2}|.(|\delta_{A_1}| + |\delta_{A_2}|))$. La complexité de l'Algorithme 2 est de l'ordre du nombre d'états de S.

[4]Soit un état s, nous définissons par $deg^o_T(s)$ le nombre de transitions qui ont s comme origine dans une série de transitions T.

6 Travaux existants

L'adaptation des composants a pour objectif essentiel de résoudre les incompatibilités lors de la réutilisation des composants préexistants. Pour développer un système sûr et compatible à base de composants, plusieurs techniques ont été proposées selon le modèle utilisé. Dans ce papier, nous nous sommes intéressés à assembler des composants d'une architecture selon leurs interfaces, qui décrivent non seulement la signatures des actions, mais également les protocoles des composants. Dans notre cas, ces protocoles sont décrits par les automates d'interface. On peut classer les techniques d'adaptation en deux catégories. Celles qui génèrent automatiquement un adaptateur entre deux composants lorsque celui-ci existe et celles qui sont semi-automatiques en proposant à l'architecte logiciel de compléter le *mapping* incomplet par des vecteurs de synchronisation. De manière formelle, l'adaptateur est issu de la composition des deux interfaces des deux composants à assembler. Dans notre cas, l'adaptateur est généré automatiquement.

Dans [18], les auteurs ont développé l'outil PaCoSuite pour modifier visuellement les composants, et de générer des adaptateurs en utilisant les signatures des méthodes et des protocoles des interfaces. Dans sa thèse de doctorat, Gschwind [7] utilise un référentiel d'adaptateurs à sélectionner de manière dynamique. Dans [5], les auteurs ont proposé une approche fondée sur l'algèbre des processus pour générer automatiquement des adaptateurs utilisant des expressions régulières et les réseaux de Petri. Dans [13] Passerone, Alfaro et Henzinger utilisent la théorie des jeux pour décider si les interfaces des composants incompatibles peuvent être rendus compatibles. Dans [9], David Hemer a proposé, à l'aide du langage CARE, des stratégies d'adaptation de composants. Dans [12], les auteurs ont proposé un modèle d'adaptateurs utilisant la méthode B, et permettant de définir l'interopérabilité entre les composants.

Dans [16], les auteurs ont montré comment utiliser les transformations de protocoles pour augmenter l'interaction des comportements d'un ensemble de composants. Cette approche porte sur les problème du renforcement des interactions entre les composants. Enfin, Bosch [4] donne un large aperçu sur les mécanismes d'adaptation, y compris celles non automatisées.

L'approche d'adaptation logicielle que nous proposons a été implantée et testée. Elle prend en considération l'adaptation aux niveaux des signatures et des protocoles. La notion de compatibilité entre interfaces et l'approche optimiste des automates d'interface sont pris en compte par notre approche.

7 Conclusion et travaux futurs

Dans cet article nous présentons une approche formelle d'adaptation de composants aux niveaux protocoles et signature des actions, entièrement automatisée et basée sur les automates d'interfaces. L'originalité de ce travail est l'exploitation de la méthode des automates des interfaces pour spécifier les interfaces des composants et pour vérifier leur compatibilité, afin de générer automatiquement des adaptateurs. Ainsi nous exploitons les points forts de l'approche des automates d'interface qui sont : un bon niveau

d'expressivité pour spécifier les protocoles des composants, et une méthode de vérification de la compatibilité efficace.

Une première étape de notre approche consiste à spécifier un mapping entre les noms d'actions de deux composants qui permet de définir des correspondances "une-pour-une", "une-pour-plusieurs" entre les actions incompatibles des deux composants. Cette fonction représente la spécification abstraite minimale de l'adaptateur. Ensuite, nous proposons un algorithme qui permet de générer un adaptateur pour deux automates d'interface adaptables en se basant sur le mapping. Actuellement, nous développons un outil pour implanter l'approche de vérification de la compatibilité entre deux composants au niveau protocole et niveau sémantique des actions [14, 15], basée sur les automates d'interface. Nous projetons aussi d'intégrer l'approche proposée dans ce papier dans l'outil.

Références

[1] L. Alfaro and T. A. Henzinger. Interface automata. *ACM Press, 9th Annual Aymposium of FSE (Foundations of Software Engineering)*, pages 109–120, 2001.

[2] L. Alfaro and T. A. Henzinger. Interface theories of component-based design. *In the proceeding of the First International Workshop of Embedded Software (EMSOFT), LNCS*, 2211:148–165, 2001.

[3] L. Alfaro and T. A. Henzinger. Interface-based design. *NATO Science Series : Mathematics, Physics, and Chemistry, Engineering Theories of Softwareintensive Systems*, 195:83Ű104, 2005.

[4] J. Bosch. Design and use of software architectures - adopting and evolving a product-line approach. *Addison-Wesley, Reading, MA, USA*, 2000.

[5] B. A. C. C. Bracciali, A. A formal approach to component adaptation. *Journal of Systems and Software*, 74:45–54, 2005.

[6] M. J. P. P. Canal, C. Software adaptation. *Special Issue on Software adaptation*, 12(1):9–31, 2006.

[7] T. Gschwind. Adaptation and composition techniques for component-based software engineering. *PhD thesis, Technische Universität of Wien*, 2002.

[8] H. G.T. An evaluation of component adaptation techniques. *In the proceeding of ICSE 99 Workshop on CBSE*, 1999.

[9] D. Hemer. A formal approach to component adaptation and composition. *In Proceedings of the Twenty-eighth Australasian conference on Computer Science ACSC '05 Newcastle, Australia*, pages 259–266, 2005.

[10] D. Konstantas. Interoperation of object oriented application. *In Proceedings of Object-Oriented Software Composition, Oscar Nierstrasz and Dennis Tsichritzis, Prentice Hall*, pages 69–95, 1995.

[11] N. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. *In the proceeding of the 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[12] L. A. Mouakher I. and S. J. Component adaptation: Specification and verification. *In 11th International Workshop on Component Oriented Programming*, 2006.

[13] d. A. L. H. T. S.-V. A. Passerone, R. Convertibility verification and converter synthesis: Two faces of the same coin. *In the Proceedings of the International Conference on Computer Aided Design (ICCAD'02)*, 2002.

[14] H. M. S. Mouelhi, S. Chouali. An i/o automata based approach to verify component compatibility: application to the cycab car. *ENTCS, FESCA08 of the European joint conference on Theory and Practice of Software (ETAPS'08)*, March 2008.

[15] H. M. S. Mouelhi, S. Chouali. Refinement of interface automata strengthened by action semantics. *ENTCS, FESCA09 of the European joint conference on Theory and Practice of Software (ETAPS'09)*, March 2009.

[16] G. D. Spitznagel, B. A compositional formalization of connector wrappers. *In the Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, Los Alamitos, CA, IEEE Computer Society*, pages 374–384, May 3-10 2003.

[17] C. Szyperski. Component software: Beyond object oriented programming. *Addison Wesley*, 1999.

[18] W. B. Vanderperren, W. Towards a new component composition process. *In the proceeding of the ECBS 2001 Int Conf, Washington, USA*, pages 322–331, 2001.

[19] P. Wegner. Interoperability. *ACM Computing Survey*, 28, 285-287 1996.