Adaptation de composants logiciels

Une approche automatisée basée sur des expressions régulières de vecteurs de synchronisation

Carlos Canal[1], **Pascal Poizat**[2], **et Gwen Salaün**[3]

[1] *University of Málaga, Department of Computer Science*
Campus de Teatinos, 29071 Málaga, Spain
email : canal@lcc.uma.es
[2] *IBISC FRE 2873 CNRS – Université d'Évry Val d'Essonne,* Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France
email : Pascal.Poizat@ibisc.univ-evry.fr
[3] *VASY project, INRIA Rhône-Alpes, France*
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
email : Gwen.Salaun@inrialpes.fr

RÉSUMÉ. *L'adaptation logicielle a pour objectif de résoudre de façon automatique et non intrusive les problèmes d'incompatibilités entre composants par la création d'entités spécifiques appelées* adaptateurs. *Si le problème des incompatibilités au niveau des signatures est partiellement supporté par les plateformes de composants industrielles (EJB, CCM) au travers notamment de l'utilisation de langages de définition d'interfaces (IDL), des incompatibilités de plus haut niveau comme celles de niveau comportemental (protocoles des composants) restent irrésolues. Dans cet article nous présentons une approche formelle d'adaptation comportementale automatique, basée sur des expressions régulières de vecteurs de synchronisation.*

ABSTRACT. *Software adaptation aims at solving automatically and in a non intrusive way mismatch between components through the use of specific entities called* adapters. *If current state-of-the-art component platforms (EJB, CCM) partially support mismatch resolution at the signature level, for example through the use of interface description languages (IDL), higher level mismatch, such as protocol ones, are still an open issue. In this article we present an automatic formal adaptation approach based on regular expressions of synchronous vectors.*

MOTS-CLÉS : *Composants logiciels, interfaces comportementales, adaptation, vecteurs de synchronisation, expressions régulières.*

KEYWORDS: *Software components, Behavioural interfaces, Adaptation, Synchronous vectors, Regular expressions.*

1. Introduction

Le génie logiciel basé composants a pour objectif de proposer un vrai marché de composants et services réutilisables, similaire en essence au marché des composants matériels et/ou électroniques. L'un des freins à son développement est que, contrairement à ce qui se passe avec les composants matériels, le logiciel est rarement réutilisé *tel quel*, mais un certain degré d'adaptation est souvent nécessaire, surtout en cas de code patrimonial (*legacy code*).

Les composants sont (ré)utilisables au travers de leurs interfaces. Celles des modèles de composants actuels (*e.g.*, EJB, CCM) ne permettent pas d'assurer la compatibilité entre eux hormis au niveau, limité, des noms et arguments de leurs services. Il est donc important, afin d'assurer l'interaction correcte des composants, que ceux-ci soient équipés d'*interfaces comportementales*. Deux autres niveaux d'interopérabilité concernent la sémantique (fonctionnalité) et la qualité de service (temps d'exécution, ressources nécessaires). L'adaptation peut avoir lieu à tous ces niveaux d'interface, où différentes incompatibilités peuvent survenir et alors devoir être corrigées.

Contrairement à l'évolution qui suppose en général l'accès au code des composants et à la *customisation* qui suppose un certain nombre de possibilités de modification prévues à l'avance, les objectifs de l'adaptation logicielle [CAN 06] sont de *produire un adaptateur* de façon *non intrusive* (pas de modification directe du code des composants qui est non accessible), le plus *automatiquement* possible (pas ou peu d'intervention humaine) et dans un contexte doublement dynamique (code de l'adaptateur non défini au départ et adaptation *at run-time*).

Nous nous intéressons ici à la construction automatique d'adaptateurs et non à la mise en œuvre du processus d'adaptation (*e.g.*, *wrappers*, réflexion, AOP, intergiciels adaptatifs). De façon abstraite, dans une approche de langages de description architecturale (ADL) un adaptateur peut être vu comme un cas particulier de connecteur dont l'objectif est d'assurer l'inter-opérabilité entre composants. Un état de l'art sur la construction d'adaptateurs [CAN 06] nous apprend que son automatisation passe par l'utilisation de formalismes dynamiques et procède en trois étapes : (i) une technique de détection d'incompatibilités est développée, (ii) une description abstraite des propriétés du système adapté, du médiateur entre composants (l'adaptateur) ou de simples correspondances entre interfaces est fournie : le *mapping d'adaptation* et (iii) le processus d'adaptation produit ensuite, automatiquement à l'aide de ce *mapping* et des interfaces des composants à adapter, l'adaptateur permettant lorsqu'il est placé au sein du contexte d'exécution d'assurer la correction du système. Le *mapping* d'adaptation peut être fourni par l'architecte logiciel. Il est aussi envisageable de l'obtenir automatiquement grâce à des techniques du domaine des services Web sémantiques [BEN 05].

Nous proposons une approche automatique outillée pour l'adaptation de composants logiciels équipés d'interfaces comportementales. Elle s'appuie sur l'utilisation conjointe de vecteurs de synchronisation et d'expressions régulières et se caractérise par les points suivants : correspondance possible entre services non uniformes (noms différents), compensation des incompatibilités comportementales, prise en compte de

la nécessité de ré-ordonnancement des appels de services, suppression des blocages. Elle constitue une avancée significative par rapport aux approches récentes d'adaptation automatique [BRA 05, BRO 06, INV 03A, INV 03B]. Plus de détails, notamment un comparatif avec ces approches, sont disponibles dans [CAN 05].

2. Description d'interfaces et test de compatibilité

Les interfaces des composants sont décrites par une signature et une description comportementale : un système de transitions étiquetées. Une *signature* Σ est un ensemble de profils de services (ou opérations) répartis en services *fournis* et *requis*. Un profil est simplement un nom de service, avec le type de ses arguments, le type du résultat et éventuellement les exceptions qu'il peut lever. Nous utilisons le terme service dans une acceptation générique (atome de base requis ou fourni par un composant). Sa granularité correspond à celle d'un échange (envoi ou réception) de message dans une approche de composants objets. Un *Système de Transitions Etiquetées* (STE) est un n-uplet (A, S, I, F, T) où : A est un alphabet (un ensemble d'événements), S est un ensemble d'états, $I \in S$ est un état initial (graphiquement dénoté comme en UML), $F \subseteq S$ sont des états finaux (graphiquement dénotés comme en UML[1]), et $T \subseteq S \times A \times S$ est une relation de transition (le STE peut être indéterministe). L'alphabet du STE est construit sur la signature. Cela signifie que pour chaque service fourni p dans une signature, il y a un élément $p?$ dans l'alphabet, et pour chaque service requis r, un élément $r!$. Etant donnée cette relation entre signature et STE, nous ne nous intéressons dans la suite qu'à ces derniers. La notation des événements correspond à l'usage dans les algèbres de processus comme LOTOS [ISO 89]. On retrouvera cependant parfois une notation différente ($?p$ au lieu de $p?$ et $!r$ au lieu de $r!$) dans la littérature. Le *mirroir* d'un événement l, noté $\bar{l}$ est défini par : $\bar{l} = e!$ si $l = e?$ et $\bar{l} = e?$ si $l = e!$.

Les STE sont une notation conviviale, mais il est parfois utile de disposer de notations plus concises comme les algèbres de processus qui sont de plus traduisibles en STE. La partie de CCS [MIL 89] réservée aux processus séquentiels peut donc aussi être utilisée comme langage de description d'interfaces comportementales : `P ::= 0 | a?.P | a!.P | P1+P2 | A`, où `0` est un processus inactif (souvent omis dans les définitions de processus), `a?.P` un processus qui reçoit `a` puis se comporte comme `P`, `a!.P` un processus qui émet `a` puis se comporte comme `P`, `P1+P2` un processus qui évolue soit en `P1` soit en `P2`, et `A` correspond à un appel de processus. Ces processus sont étiquetés par les symboles `[i]` et `[f]` pour représenter les états initiaux et finaux.

Exemple 1. Considérons un système clos avec un client qui reçoit en boucle une requête et ses arguments, et attend une réponse, terminant avec `end!` ainsi qu'un serveur qui reçoit des requêtes avec valeurs, puis fournit un service.

1. Contrairement à UML où les états finaux sont des pseudo-états, le caractère final est ici un attribut complémentaire des états normaux.

```
Client[i] = query!.arg!.ack?.Client + end![f]
Server[i,f] = query?.value?.service!.Server.
```

Les STE correspondants sont donnés dans la figure 1.

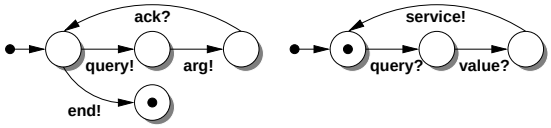

Figure 1. *STE des composants de l'exemple 1*

Différentes notions de compatibilité de comportements ont été proposées dans le cadre des composants [CAN 06]. Les plus communément acceptées correspondent à l'absence de blocage. Soit $L = (A, S, I, F, T)$ un STE. Un état s est un *état de blocage* (deadlock) pour L, noté $\texttt{dead}(s)$, s'il appartient à S, qu'il n'a pas de transitions sortantes, mais qu'il n'appartient pas à $F : s \in S \wedge s \notin F \wedge \nexists l \in A, s' \in S \,.\, (s, l, s') \in T$. Un STE présente une *incompatibilité (comportementale)* s'il a au moins un état s tel que $\texttt{dead}(s)$. Dans l'exemple 1, un blocage provient du fait que (i) le service requis `end !` dans le client n'a pas d'équivalent chez le serveur (on suppose que tout appel doit être acquité), et (ii) que les noms des services `arg !` et `value ?` respectivement du côté du client et du serveur ne correspondent pas.

3. Notations, algorithmes et outils pour l'adaptation logicielle

Nous vérifions l'absence d'incompatibilité comportementale en rajoutant des boucles `accept` sur les états finaux des interfaces, en calculant leur produit synchrone, puis en vérifiant l'absence de blocage sur ce produit. Ces opérations sont automatisées [CAN 05] par la transformation des interfaces des composants (après ajout des boucles `accept`) au format `.aut` (format textuel d'automate accepté en entrée de CADP [GAR 02]), puis par la création d'une description de l'architecture des composants en script pour EXP.OPEN [LAN 05] qui s'en sert pour calculer le produit.

Un adaptateur est décrit par un STE qui, ajouté au sein d'un système contenant des incompatibilités, permet d'obtenir un système qui n'en contient plus. Afin que cela fonctionne, tous les appels de services doivent être préemptés par l'adaptateur. Formellement, avant de débuter le processus d'adaptation, les noms des services doivent être préfixés par le nom du composant, *e.g.*, `c:service !`.

Nous donnons dans la figure 2, une vue d'ensemble de notre approche. Les interfaces de composants sont utilisées pour vérifier une possible incompatibilité. Si c'est le cas, alors l'adaptation doit être appliquée. Deux niveaux d'adaptation comportementale sont possibles : avec ou sans ré-ordonnancement. Dans les deux cas, notre procédure permet de définir abstraitement les propriétés de l'adaptateur (ou du système adapté final). La notation proposée s'appuie sur les vecteurs de synchronisation et les expressions régulières.

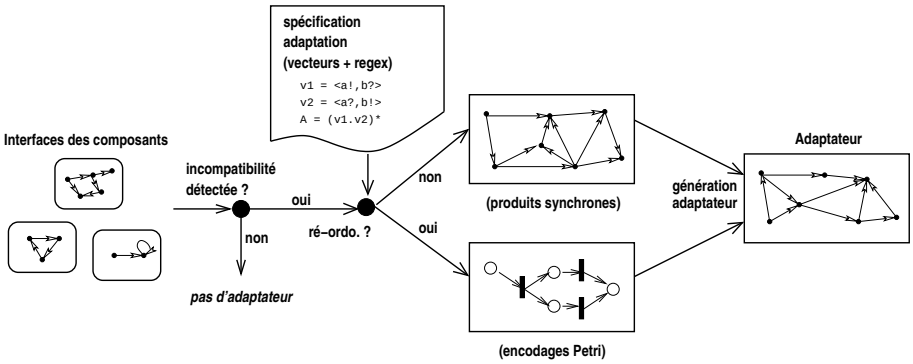

Figure 2. *Vue d'ensemble de notre approche d'adaptation*

Un *vecteur synchrone* (ou plus simplement *vecteur*) pour un ensemble de composants $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in Id$, est un n-uplet (e_i) avec $e_i \in A_i \cup \{\varepsilon\}$, ε exprimant qu'un composant ne participe pas à une synchronisation. Les vecteurs n'expriment pas seulement la synchronisation entre processus sur les mêmes noms d'événements, mais permettent plus généralement des correspondances entre différents noms d'événements.

Étant donné n STE $L_i = (A_i, S_i, I_i, F_i, T_i)$, et un ensemble de vecteurs $V = \{(e_{ij})\}_j$ avec $e_{ij} \in A_i \cup \{\varepsilon\}$, une expression régulière de vecteurs pour ces STE peut être engendrée à partir de la syntaxe suivante : *R ::= v (VECTEUR)| R1.R2 (SÉQUENCE)| R1+R2 (CHOIX)| R* (ITÉRATION)*, où R, R1, R2 sont des expressions régulières et v est un vecteur de V. Une description graphique (STE étiqueté par des vecteurs) peut être utilisée à la place des expressions régulières afin de favoriser la lisibilité et convivialité de la notation.

Étant donnés un ensemble de descriptions d'interfaces de composants et une spécification abstraite de l'adaptation souhaitée, un adaptateur peut être engendré automatiquement. Nous donnons ici les principes de notre approche ainsi que l'algorithme pour l'adaptation avec ré-ordonnancement. Les algorithmes liés à l'adaptation sans ré-ordonnancement pourront être trouvés dans [CAN 05].

Dans le cas de l'**adaptation sans ré-ordonnancement**, un produit synchrone est obtenu à partir d'un STE encodant l'expression régulière du *mapping* et des miroirs (en termes d'entrées/sorties) des interfaces des composants. Le STE produit est étiqueté par des vecteurs. Il est passé à un script qui y supprime les chemins permettant d'atteindre un état de blocage ; une alternative est de faire un retour à l'architecte logiciel qui peut alors proposer une mise à jour du *mapping*. Enfin les différents entrelacements possibles pour chaque synchronisation décrite par un vecteur (tous les scénarios possibles de communication entre les composants et l'adaptateur qui implante ce vecteur) permettent d'engendrer le STE de l'adaptateur.

Dans le cas de l'**adaptation avec ré-ordonnancement** d'appels de services, l'idée est d'encoder dans un réseau de Petri l'ensemble des contraintes liées à l'adaptateur. Des places-ressources y correspondent aux différents services (places notées avec `??` ou `!!`) ainsi qu'aux états des interfaces de composants. Les miroirs (en termes d'entrées/sorties) des interfaces des composants, les relations de correspondance induites par les vecteurs et les restrictions sur les enchaînements possibles de ces correspondances induites par l'expression régulière du *mapping* permettent la construction des transitions et des arcs du réseau.

L'algorithme détaillé est le suivant :

1) pour chaque composant i ayant pour STE $L_i = (A_i, S_i, I_i, F_i, T_i)$, pour chaque état $s_j \in S_i$, ajouter une place `Control-i-s_j`

2) pour chaque composant i ayant pour état initial I_i, ajouter un jeton dans la place `Control-i-I_i`

3) pour chaque $a!$ dans $\bigcup_i A_i$, ajouter une place `??a`

4) pour chaque $a?$ dans $\bigcup_i A_i$, ajouter une place `!!a`

5) pour chaque composant i ayant pour STE L_i, pour chaque $(s, l, s') \in T_i$:

a) ajouter une transition ayant pour étiquette $\bar{l}$, un arc de la place `Control-i-s` à la transition et un arc de la transition à la place `Control-i-s'`

b) si `l` est de la forme $a!$ alors ajouter un arc de la transition à la place `??a`

c) si `l` est de la forme $a?$ alors ajouter un arc de la place `!!a` à la transition

6) calculer le STE $L_R = (A_R, S_R, I_R, F_R, T_R)$ correspondant à l'expression régulière R [HOP 79]

7) pour chaque état s_R de S_R, ajouter une place `Control-R-s_R`

8) ajouter un jeton dans la place `Control-R-I_R`

9) pour chaque transition $t_R = (s_R, (l_1, \ldots, l_n), s'_R)$ de T_R :

a) ajouter une transition ayant pour étiquette `tau`, un arc de la place `Control-R-s_R` à la transition et un arc de la transition à la place `Control-R-s'_R`

b) pour chaque l_i de la forme $a!$, ajouter un arc de la place `??a` à la transition

c) pour chaque l_i de la forme $a?$, ajouter un arc de la transition à la place `!!a`

10) pour chaque n-uplet $(f_1, \ldots, f_n, f_R)$, $f_i \in F_i$, $f_R \in F_R$, d'états finaux, ajouter une transition (boucle) `accept` avec des arcs de et vers chacun des éléments du n-uplet.

Le STE de l'adaptateur est ensuite obtenu par graphe de marquage (adaptateurs non récursifs) ou de couverture (adaptateurs récursifs) du réseau. Cette approche se base sur l'utilisation de scripts de traduction et d'encodage des contraintes en réseau de Petri, sur l'outil TINA [BER 04] qui permet l'obtention des graphes de marquage et de couverture et enfin sur l'outil CADP qui permet différentes simplifications com-

portementales sur ces graphes (comme par exemple les réductions observationnelles supprimant les τ-transitions, notre processus en engendrant quantité[2]).

Exemple 2. Supposons un nouveau système composé d'un client `c` et de deux serveurs `s` et `a` :

```
C[i] = c:end![f] + c:req!.c:arg!.c:ack?.C
S[i,f] = s:value?.s:query?.s:service!.S
A[i,f] = a:value?.a:query?.a:service!.A
```

Une possible adaptation pourrait obliger le client à accéder aux serveurs alternativement. Dans cette optique l'expression régulière suivante peut être utilisée :
$R_1 = (\mathtt{v}_{s1}.\mathtt{v}_{s2}.\mathtt{v}_{s3}.\mathtt{v}_{a1}.\mathtt{v}_{a2}.\mathtt{v}_{a3})*.\mathtt{v}_{\mathrm{end}}$ avec

$\mathtt{v}_{s1} = <\mathtt{c:req!},\mathtt{s:query?},\varepsilon>$, $\mathtt{v}_{a1} = <\mathtt{c:req!},\varepsilon,\mathtt{a:query?}>$,
$\mathtt{v}_{s2} = <\mathtt{c:arg!},\mathtt{s:value?},\varepsilon>$, $\mathtt{v}_{a2} = <\mathtt{c:arg!},\varepsilon,\mathtt{a:value?}>$,
$\mathtt{v}_{s3} = <\mathtt{c:ack?},\mathtt{s:service!},\varepsilon>$, $\mathtt{v}_{a3} = <\mathtt{c:ack?},\varepsilon,\mathtt{a:service!}>$,
$\mathtt{v}_{end} = <\mathtt{c:end!},\varepsilon,\varepsilon>$

Une adaptation moins contraignante serait de laisser le choix au client d'accéder au serveur qu'il désire : $R_2 = (\mathtt{v}_{s1}.\mathtt{v}_{s2}.\mathtt{v}_{s3} + \mathtt{v}_{a1}.\mathtt{v}_{a2}.\mathtt{v}_{a3})*.\mathtt{v}_{\mathrm{end}}$.

L'application de notre démarche (figure 3) à R_2 produit d'abord un réseau de Petri (à gauche). Pour des raisons de lisibilité, nous avons omis l'encodage du composant `a`, le nommage des places de contrôle et nous n'avons indiqué les arcs des transferts de services que pour un seul vecteur, $\mathtt{v}_{s1}$. L'adaptateur correspondant (à droite) résoud l'incompatibilité initiale (plus de blocages) mais assure aussi le respect des contraintes spécifiées par R_2. L'utilisation d'expressions régulières est donc un moyen d'imposer des contraintes temporelles sur le système sous adaptation.

4. Conclusion

Dans cet article, nous avons présenté notre approche pour l'adaptation logicielle. Elle combine les intérêts des approches récentes d'adaptation automatique outillée [BRA 05, BRO 06, INV 03A, INV 03B]. Ses points principaux sont : (i) un bon niveau d'expressivité fourni par notre langage de description abstraite de l'adaptateur (vecteurs et expressions régulières), et (ii) des mécanismes d'adaptation riches permettant par exemple le ré-ordonnancement. Elle est outillée à l'aide de scripts écrits en python et d'interfaces avec les outils CADP et TINA.

Une perspective en cours est la liaison avec un modèle de composants (comme par exemple l'implantation de Fractal en Proactive [BAU 03]) dans le but d'engendrer automatiquement des composants adaptateurs dans une architecture de façon dynamique. Une autre perspective est l'extension de l'approche à la prise en compte de

2. τ- ou `tau`-transitions, transitions internes modélisant explicitement dans notre approche le transfert des appels de service.

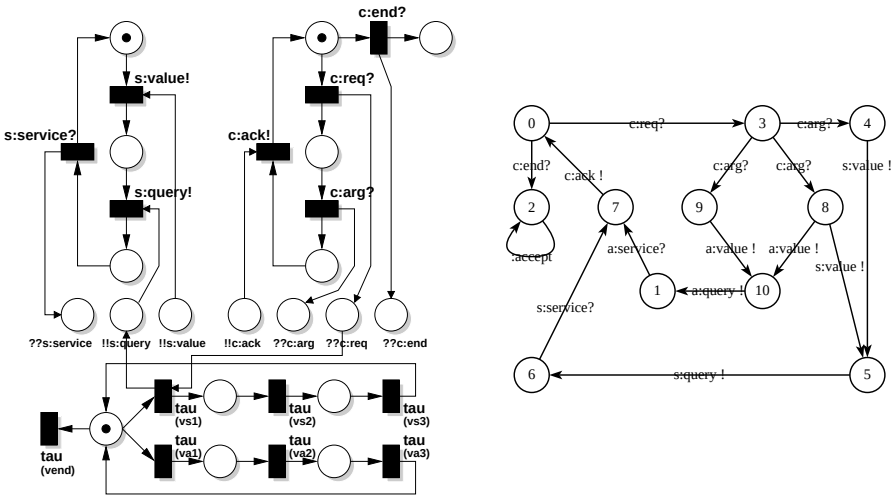

Figure 3. *Résultats de l'adaptation avec ré-ordonnancement du système client/serveur*

données et d'informations de qualité de service temporelles dans les interfaces et dans les *mappings* en utilisant les capacités temporelles de TINA. Enfin, nous voulons appliquer notre approche à une étude de cas de taille significative.

Bibliographie

[BAU 03] BAUDE F., CAROMEL D., MOREL M., « From Distributed Objects to Hierarchical Grid Components », *Distributed Objects and Applications*, Springer Verlag, LNCS 2888, 2003, p. 1226-1242.

[BEN 05] BEN MOKHTAR S., GEORGANTAS N., ISSARNY V., « Ad Hoc Composition of User Tasks in Pervasive Computing Environments », *Software Composition*, Springer Verlag, LNCS 3628, 2005, p. 31-46.

[BER 04] BERTHOMIEU B., RIBET P.-O., VERNADAT F., « The Tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », *International Journal of Production Research*, vol. 42, n° 14, 2004.

[BRA 05] BRACCIALI A., BROGI A., CANAL C., « A Formal Approach to Component Adaptation », *Journal of Systems and Software*, vol. 74, n° 1, 2005, p. 45-54.

[BRO 06] BROGI A., CANAL C., PIMENTEL E., « Component Adaptation Through Flexible Subservicing », *Science of Computer Programming*, À paraître.

[CAN 05] CANAL C., POIZAT P., SALAÜN G., « Adaptation of Component Behaviours using Synchronous Vectors », Technical Report ITI-05-10, Université de Málaga, 2005. Disponible sur le site web de P. Poizat.

[CAN 06] CANAL C., MURILLO J. M., POIZAT P., « Software Adaptation », *L'Objet. Special Issue on Coordination and Adaptation Techniques*, vol. 12, n° 1, 2006, p. 9-31.

[GAR 02] GARAVEL H., LANG F., MATEESCU R., « An Overview of CADP 2001 », *EASST Newsletter*, vol. 4, 2002, p. 13-24.

[HOP 79] HOPCROFT J. E., ULLMAN J. D., « Introduction to Automata Theory, Languages and Computation », *Addison Wesley*, 1979.

[INV 03a] INVERARDI P., TIVOLI M., « Deadlock Free Software Architectures for COM/DCOM Applications », *Journal of Systems and Software*, vol. 65, n° 3, 2003, p. 173-183.

[INV 03b] INVERARDI P., TIVOLI M., « Software Architecture for Correct Components Assembly », Chapitre 6 de *Formal Methods for the Design of Computer, Communication and Software Systems : Software Architecture*, Springer Verlag, LNCS 2804, 2003, p. 92-121.

[ISO 89] ISO, « LOTOS : a Formal Description Technique based on the Temporal Ordering of Observational Behaviour », Technical Report 8807, International Standards Organisation, 1989.

[LAN 05] LANG, F., « EXP.OPEN 2.0 : A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verifi cation Methods », *Integrated Formal Methods*, Springer Verlag, LNCS 3771, 2005, p. 70-88.

[MIL 89] MILNER R., « Communication and Concurrency », *Prentice Hall*, International Series in Computer Science, 1989.