
Contrats comportementaux pour un assemblage sain de composants

Cyril Carrez* — Alessandro Fantechi** — Elie Najm*

* *Ecole Nationale Supérieure des Télécommunications, Département INFRES*
46 rue Barrault,
F-75013 Paris, France
{cyril.carrez, elie.najm}@enst.fr

** *Università di Firenze, Dipartimento di Sistemi e Informatica*
Via S. Marta 3,
I-50139 Firenze - Italy
fantechi@dsi.unifi.it

RÉSUMÉ. La conception basée composants est une nouvelle méthode de construction d'applications et de systèmes distribués. La vérification compositionnelle de ces systèmes reste cependant un problème. Nous étudions des méthodes et des concepts pour la construction d'assemblages "sains". Nous définissons un modèle de composant abstrait, dynamique et multi-tâche, qui englobe les modèles client/serveur et point-à-point (peer to peer). Nous définissons un langage de type d'interfaces comportementales, doté d'un ensemble de règles (décidables) de compatibilité d'interface. Basé sur la notion de conformité du composant par rapport à son interface, nous définissons les concepts de "contrat" et "respect de contrat". Cela mène au concept d'assemblages sains de composants : assemblages faits de composants et leurs contrats, interagissant à travers des interfaces compatibles. Les assemblages sains possèdent des propriétés intéressantes, comme "l'absence d'interblocage externe" et "la consommation des messages".

ABSTRACT. Component based design is a new methodology for the construction of distributed systems and applications. Remains the problem of the compositional verification of such systems. We investigate methods and concepts for the provision of "sound" assemblies. We define an abstract, dynamic, multi-threaded, component model, encompassing both client/sever and peer to peer communication patterns. We define a behavioural interface type language endowed with a (decidable) set of interface compatibility rules. Based on the notion of compliance of components to their interfaces, we define the concepts of "contract" and "contract satisfaction". This leads to the concept of sound assemblies of components, i.e., assemblies made of contracted components interacting through compatible interfaces. Sound assemblies possess interesting properties, such as, "external deadlock freeness" and "message consumption".

MOTS-CLÉS: Typage comportemental, vérification, composition, composants, peer-to-peer, client/serveur

KEYWORDS: Behavioural typing, verification, composition, components, peer-to-peer, client/server

1. Introduction

Les systèmes de type comportementaux ont été définis ces dernières années dans le but d'être capable de vérifier la compatibilité d'objets communicants et concurrents, non seulement vis-à-vis des données échangées, mais aussi vis-à-vis des correspondances entre leur comportement respectif [NIE 95, KOB 99, NAJ 99]. Ce contrôle trouve naturellement une application dans la vérification de la compatibilité des composants, d'autant plus que les développements récents dans l'ingénierie du logiciel se situent autour de la conception basée composants (un logiciel est construit à partir de composants métiers existants, qui sont connectés entre eux soit par du code spécifique, soit en ayant recours à une plate-forme standard comme CORBA, .NET). La compatibilité d'un composant avec l'environnement dans lequel il va être utilisé doit être garantie avant son déploiement.

Les techniques de vérification formelle peuvent donc jouer un rôle stratégique dans le développement de logiciels de qualité : dans l'idée des *méthodes formelles légères*, le concepteur d'application, s'il se contente d'assembler des composants, ne s'occupe pas de la description formelle du logiciel qu'il développe, mais a la garantie de l'absence d'incompatibilité entre les composants, à partir d'une vérification formelle sous-jacente des composants et d'une vérification formelle contrôlant la compatibilité des types. Un exemple encore plus frappant est le code mobile, où il faut absolument garantir qu'un composant mobile ne trouble pas le comportement correct du composant destination. Cette vérification doit avoir lieu à l'exécution, à la réception du composant mobile, et doit donc être faite avec efficacité. Pour cette raison, nous devons considérer non seulement la fonctionnalité du composant en entier, mais aussi une abstraction adéquat de son comportement, qui sera suffisante pour prouver que des propriétés de la configuration globale de composants ne sont pas altérées par la composition.

Dans ce travail, nous définissons un cadre dans lequel un composant affiche plusieurs *interfaces* à travers lesquelles il communique. A chaque interface est associé un type, qui est une abstraction du comportement du composant. L'utilisation de préfixes **must** et **may** permet la distinction entre respectivement les messages *requis* et les messages *possibles*. La complexité du langage de type d'interface est volontairement faible, pour faciliter la vérification de la compatibilité entre interfaces. Nous ne donnons pas de langage spécifique pour les composants, mais plutôt une définition abstraite qui se veut suffisamment générale pour être adaptée aux différents langages. Les composants sont vus comme un ensemble de ports, à travers lesquels ils communiquent, et un ensemble de tâches (ou *thread*) dans lesquelles nous n'observons que les effets sur les ports. Sous certaines contraintes sur l'utilisation des ports à l'intérieur du composant, nous montrons qu'une configuration faite de composants communicants est bien typée et vérifie des propriétés de vivacité, si le composant respecte le contrat donné par ses interfaces, et que les interfaces communicantes sont compatibles.

Notre travail est en partie inspiré par De Alfaro et Henzinger [ALF 01], qui associent des automates d'interface aux composants et définissent des règles de compatibilité entre interfaces. Notre approche, qui se situe plutôt dans la lignée des systèmes de type d'algèbres de processus, apporte la compatibilité entre composants et interfaces : une interface est vue comme un *contrat* avec l'environnement extérieur que le composant doit honorer. En même temps, notre but est de limiter au maximum la complexité de la vérification

de la compatibilité des interfaces, qui peut ainsi être lancée à l'exécution. Le travail sur les Systèmes de Transition Modaux par Larsen, Steffen et Weise [LAR 95] a inspiré notre définition des modalités, et la façon dont la compatibilité des interfaces est contrôlée. La garantie de satisfaction des propriétés de sûreté de typage et de vivacité a été couverte par Najm, Nimour et Stefani [NAJ 99], et nous avons hérité de leur approche en montrant comment la vérification des règles de compatibilité garantit des propriétés plus générales.

L'article est structuré comme suit : la section 2 expose le modèle de composant sur lequel nous basons la définition (section 3) de notre langage d'interface et les règles de compatibilité correspondantes. Dans la section 4, nous donnons le concept de *composant respectant un contrat*. La section 5 décrit les propriétés qui sont garanties dans ce cadre.

2. Modèle de composant

2.1. Présentation informelle

Notre modèle de composant décrit un système comme une configuration de composants communicants. Chaque composant possède un ensemble de ports, et la communication se fait par envoi asynchrone de messages entre les ports. L'envoi de message ne peut avoir lieu que si le port est attaché à un autre port "partenaire". Si un port est attaché, tout message envoyé par ce port est dirigé vers le port partenaire. Par contre, un port qui n'est pas attaché ne peut qu'effectuer des réceptions. Nous considérons des configurations dynamiques : un composant peut créer de nouveaux ports, et dynamiquement attacher une référence d'un port partenaire à l'un de ses ports (par contre, la création dynamique de composant n'est pas encore prise en compte). Dans notre environnement, les communications de type client/serveur et point-à-point (*peer-to-peer*) peuvent être modélisées. Lorsque deux ports sont attachés l'un à l'autre, ils sont pairs (*peers*). Lorsque l'attachement est asymétrique, le port attaché à un autre est le client, et celui qui n'est pas attaché est le serveur. La figure 1 présente une configuration de trois composants. On remarque que le port c de C_1 est attaché de manière asymétrique au port s de C_2 (l'étiquette $*$ indique ici que la référence s est un serveur), et la liaison point-à-point entre les ports x et y (respectivement de C_2 et C_3) et les ports u et v (tous deux de C_1).

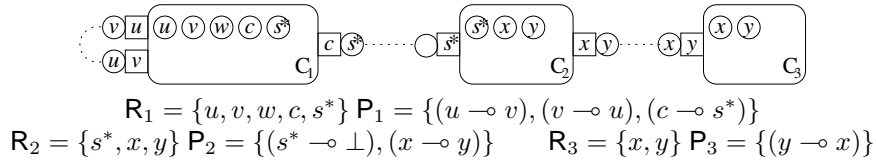


Figure 1. Un exemple de configuration

De plus, Les composants sont multi-tâches. Nous considérons ici un modèle de tâches abstraites, en se concentrant uniquement sur les exécutions externes portant sur les ports. Ainsi, une tâche active est une chaîne faite d'un port de tête (le port actif), et d'une queue (une séquence ordonnée de ports suspendus). Cette chaîne peut être augmentée (ou diminuée) dynamiquement : par exemple, elle augmente lorsque le port de tête est suspendu et l'activité est donnée à un autre port.

Comme cet article se concentre sur les problèmes de typage d'interface, nous ne donnerons pas de syntaxe complète pour les composants. Nous définissons plutôt un modèle abstrait de comportement du composant, suivant ses transitions observables, et les activités multi-tâches sur ses ports. Le modèle abstrait défini dans cette section est général et indépendant de toute notation concrète de comportement pour les composants.

2.2. Notations pour les Composants

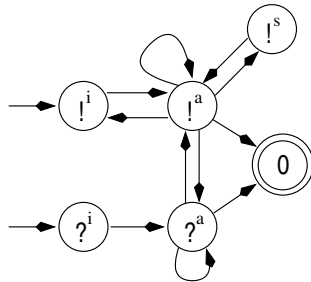
Un composant comporte un état, un ensemble de ports, un ensemble de références (l'union des partenaires et des ports), et une collection de tâches.

L'ensemble des références est noté R , ses éléments sont notés u, v, w, c, s . La notation ensembliste classique est utilisée pour les opérations sur R ; cependant, nous utilisons la notation abusive $R \cup u$ pour l'insertion d'un élément.

L'ensemble des ports est noté P . Un élément $(\cdot \multimap \cdot)$ de P est, en fait, une correspondance d'une référence de port vers une référence de partenaire. Par exemple, $(u \multimap v)$ indique que le partenaire v est attaché au port local u – un port u qui n'a pas de partenaire est noté $(u \multimap \perp)$. Les notations suivantes seront utilisées pour la manipulation de P :

$P[u \multimap \perp]$	le port u est rajouté à P .
$P[u \multimap v]$	attache le partenaire v au port u . Efface le partenaire précédent.
$P \setminus u$	retire le port u de P .
$(u \multimap v) \in P$	le port u est dans P , et attaché à v . Nous écrirons aussi $u \in P$ pour vérifier l'appartenance de u à P .

L'ensemble des tâches, T , reflète les états des ports du composant et les dépendances entre ces ports. L'état d'un port présent dans P est à la fois une abstraction de : activé, suspendu ou oisif (*idle*), et une de : envoi, réception ou pas d'action. Formellement, nous notons l'état d'activité d'un port u par $u\rho^\sigma$, avec ρ et σ explicités ci-après :



$$\rho = \begin{cases} ! & u \text{ est dans un état d'envoi} \\ ? & u \text{ est dans un état de réception} \\ 0 & u \text{ n'a pas d'action} \end{cases}$$

$$\sigma = \begin{cases} a & u \text{ est actif} \\ s & u \text{ est suspendu (par un autre port)} \\ i & u \text{ est oisif (idle)} \end{cases}$$

Le diagramme d'état indique les évolutions possibles de l'activité d'un port. On remarque que la combinaison $u?^s$ est interdite, ainsi un port en attente de réception est toujours actif; $u?^a$ est un port actif en attente de message. Un port $u!^a$ peut soit envoyer un message, soit être suspendu par un autre port. Enfin, la terminaison d'un port est représentée par 0^σ .

Les états des ports seront notés x, y . Nous définissons aussi $x \multimap y$ qui indique que x est suspendu par y . Cela veut dire que l'activité de x est suspendue jusqu'à ce que y termine

ou devienne oisif. $T = t_1 | \dots | t_n$ est un ensemble de tâches parallèles où une tâche t est une séquence $x_1 \rightsquigarrow \dots \rightsquigarrow x_n$. Cette séquence a les contraintes suivantes :

$$\begin{aligned} x_i &= u_i!^s \text{ ssi } i < n && \text{(tous les ports, excepté le dernier, sont suspendus)} \\ n > 1 \Rightarrow x_n &= u_n\rho_n^a && \text{(une séquence de plusieurs ports se termine par un port actif)} \end{aligned}$$

Les opérations sur T sont définies ci-après (avec $x = u\rho^\sigma$ présent au plus fois dans T) :

$T x$	ajoute un port qui a sa propre tâche d'exécution (pas de dépendance).
$T \setminus u$	cette opération n'est définie que si u est la tête d'une tâche $t \in T$. Elle retire le port u de t , et active le port qui précédait u .
$T[u \rightsquigarrow v]$	cette opération n'est définie que si u est la tête d'une tâche $t_1 \in T$, et v est dans une tâche singleton $t_2 = v\rho^j \in T$. Elle change l'état de u à suspendu, ajoute $v\rho^a$ comme nouvelle tête de t_1 , et retire t_2 de T .
$T[u\rho'/u\rho]$	modifie l'état d'un port u : seul ρ change en ρ' .
$T[u\rho^\sigma \rightarrow \rho']$	change l'activité d'un port.
$T(u)$	retourne ρ^σ si $u\rho^\sigma \in T$.

2.3. Médium de Communication

Comme indiqué dans l'introduction, la communication entre les composants se fait par messages asynchrones. Ainsi, un message est d'abord déposé par l'émetteur dans un médium de communication, et, un peu plus tard, retiré de ce médium par le récepteur. Nous définissons Com , une abstraction du médium de communication, contenant une liste de files FIFO (*First In, First Out*), une pour chaque référence de port contenue dans le composant : les messages sont écrits et lus à partir de Com . Nous adoptons les notations :

$Com[\triangleleft u]$	insère une nouvelle file dans Com , pour la référence u .
$Com.u$	la file pour la référence u dans Com . C'est une suite ordonnée de messages de la forme $v : M(\tilde{w})$ où v est la référence du port émetteur, M est le nom du message, et \tilde{w} ses arguments. Si u n'est pas encore défini, $Com.u = \perp$.
$Com \setminus u$	la file de u est retirée de Com .
$Com[u \triangleright]$	retire, dans la file u , le message se trouvant au dessus de la pile.
$Com[u \triangleleft v : M(\tilde{w})]$	ajoute le message $v : M(\tilde{w})$ dans la file associée à u .
$Com.u \triangleright$	donne le prochain message (en haut de la file u) à être traité.

2.4. Sémantique des composants

Un composant est défini par : $C = B(P, R, T)$, où B est l'état du composant, et P , R , T sont les ports, références et tâches comme définis précédemment.

Les règles du tableau 1 décrivent la sémantique des composants, montrant les transitions qu'un composant peut effectuer avec une abstraction de communication donnée. Une transition peut changer l'état du composant lui-même et/ou celui de l'abstraction de communication. Les deux premières règles décrivent les relations entre le composant et Com , en ce qui concerne les envois et réceptions de messages : le message est déposé, ou retiré, de la file d'attente correspondante. Les règles CCREAT et CREMV donnent la création et la suppression d'un port (qui implique la création/suppression de la file d'attente correspondante dans Com). CBIND et UNBIND sont utilisées respectivement pour l'attachement et

le détachement d'une référence partenaire à un port, liant ainsi un port partenaire à un port local. Enfin, CACTV et CACTV2 décrivent comment un port v est activé, respectivement lorsque u est suspendu par v , et lorsque v a sa propre tâche d'exécution. DEACTV permet la désactivation d'un port (le rendant oisif).

2.5. Configuration de Composants

Lorsque nous prenons en compte une configuration faite de plusieurs composants, nous considérons que le médium de communication Com est partagé entre tous les composants. Ainsi, les files d'attente sont partagées et les composants peuvent communiquer entre eux. Le tableau 2 donne la règle de communication pour une configuration de deux composants ; l'extension à une configuration à plus de deux composants est immédiate.

3. Types d'interface

Dans cette section nous décrivons le langage utilisé pour définir les interfaces. Un composant typé est un composant pour lequel toute référence initiale a un type associé, et toute création de référence ou réception de référence a aussi un type déclaré. Nous adoptons un langage de type comportemental ([NIE 95, KOB 99, NAJ 99]). Dans cet environnement, le type d'une référence décrit ses états possibles, et pour chaque état, les actions requises et/ou permises à travers cette référence, et les états après l'exécution d'une action. Une des principales caractéristiques de ce langage est l'utilisation de modalités **may** et **must**.

3.1. Syntaxe du langage d'interface

La syntaxe du langage d'interface est donné dans le tableau 3. Les mot-clés **!** et **?** correspondent aux actions habituelles d'envoi et de réception. Les modalités **may** et **must** distinguent la permission de l'obligation d'effectuer une action. L'opérateur de choix **+** permet de choisir un message parmi d'autres, et le **;** est utilisé pour séquencer les comportements. La signification des modalités est la suivante :

- may ?** ΣM_i "le port n'impose aucune contrainte d'envoi sur le partenaire ; si le partenaire envoie un message M_i , le port garantit qu'il est prêt à le recevoir".
- must ?** ΣM_i "le port impose une contrainte d'envoi sur le partenaire ; si le partenaire envoie un message M_i , le port garantit qu'il est prêt à le recevoir".
- may !** ΣM_i "le port peut envoyer au partenaire un des messages M_i , et le partenaire doit être prêt à le recevoir".
- must !** ΣM_i "le port garantit qu'il enverra un des messages M_i à son partenaire, et que le partenaire doit être prêt à le recevoir".

Les messages contiennent des arguments. Ainsi, les références vers les ports peuvent être envoyés dans les messages. Notre langage de type ne traite pas les types basiques (comme les entiers, réels, ...), mais leur ajout est immédiat. L'envoi ou la réception de références suppose quelques restrictions quant au comportement des composants impliqués :

- ! m (I)** "le port envoie une référence vers un port dont le comportement est décrit par le type I . De plus, la première action de ce port référencé doit être **?**".
- ? m (I)** "le port reçoit une référence vers un autre port dont le comportement est décrit par I . La première action de ce port référencé est **?**".

$$\begin{array}{c}
 \text{CSEND} \frac{R' \subseteq R \quad T' = T[u! / u\rho] \quad Com' = Com[u' \triangleleft u : M(\tilde{v})]}{B(P, R, T), Com \xrightarrow{u:u'!M(\tilde{v})} B'(P, R', T'), Com'} \triangle \\
 \\
 \text{CRECV} \frac{\begin{array}{l} P' = P[u \multimap u'] \quad R' \subseteq R \cup \{\tilde{v}, u'\} \quad T' = T[u? / u\rho] \\ Com' = Com[u \triangleright] \end{array}}{B(P, R, T), Com \xrightarrow{u:u'?M(\tilde{v})} B'(P', R', T'), Com'} \nabla \\
 \\
 \text{CCREAT} \frac{\begin{array}{l} P' = P[u \multimap \perp] \quad R' = R \cup u \\ T' = T | u\rho^i \quad Com' = Com[\triangleleft u] \end{array}}{B(P, R, T), Com \rightarrow B'(P', R', T'), Com'} u \notin P \wedge Com.u = \perp \\
 \\
 \text{CREMV} \frac{P' = P \setminus u \quad R' = R \setminus u \quad T' = T \setminus u \quad Com' = Com \setminus u}{B(P, R, T), Com \rightarrow B'(P', R', T'), Com'} \square \\
 \\
 \text{CBIND} \frac{P' = P[u \multimap v]}{B(P, R, T), Com \rightarrow B'(P', R, T), Com} v \in R \wedge (u \multimap \perp) \\
 \\
 \text{CUNBIND} \frac{P' = P[u \multimap \perp]}{B(P, R, T), Com \rightarrow B'(P', R, T), Com} (u \multimap v) \wedge T(u) = \rho^i \\
 \\
 \text{CACTV} \frac{T' = T[u \multimap v]}{B(P, R, T), Com \rightarrow B'(P, R, T'), Com} \diamond \\
 \\
 \text{CACTV2} \frac{T' = T[u\rho^{i \rightarrow a}]}{B(P, R, T), Com \rightarrow B'(P, R, T'), Com} T(u) = \rho^i \\
 \\
 \text{CDEACT} \frac{T' = (T \setminus u) | u\rho^i}{B(P, R, T), Com \rightarrow B'(P, R, T'), Com} T(u) = \rho^a \wedge \rho \neq ?
 \end{array}$$

$$\triangle \triangleq (u \multimap u') \in P \wedge T(u) = \rho^a \wedge \tilde{v} \subseteq R \wedge (\forall v \in \tilde{v} \cap P : T(v) = \rho^a) \wedge u' \in Com$$

$$\nabla \triangleq u \in P \wedge Com.u \triangleright = u' : M(\tilde{v}) \wedge T(u) = \rho^a$$

$$\square \triangleq Com.u = \emptyset \wedge T(u) = \rho^i \quad \diamond \triangleq T(u) = \rho^a \wedge T(v) = \rho^i \wedge (v \neq \perp)$$

Tableau 1. Règles pour la sémantique de composants

$$\text{CPAR} \frac{B_1(P_1, R_1, T_1), Com \xrightarrow{\alpha} B'_1(P'_1, R'_1, T'_1), Com'}{B_1(P_1, R_1, T_1) | B_2(P_2, R_2, T_2), Com \xrightarrow{\alpha} B'_1(P'_1, R'_1, T'_1) | B_2(P_2, R_2, T_2), Com'}$$

Tableau 2. Règles pour la configuration de composants

<i>type</i>	::=	<i>server_name</i> = <i>mod receive</i> *
		<i>peer_name</i> = (<i>mod send</i> <i>mod receive</i>)
<i>send</i>	::=	! [$\sum_i M_i$; <i>I</i>]
<i>receive</i>	::=	? [$\sum_i M_i$; <i>I</i>]
<i>I</i>	::=	0 <i>peer_name</i> <i>mod send</i> <i>mod receive</i>
<i>mod</i>	::=	may must
<i>M</i>	::=	<i>name</i> (\widetilde{args})
<i>args</i>	::=	<i>peer_name</i> <i>server_name</i> *

Tableau 3. Syntaxe BNF du langage d'interface

Enfin, la construction * permet la spécification d'un serveur :

$I = \text{mod ? } [m() ; I']^*$ après la réception de m , le port dont le comportement est I' est créé, tandis que le serveur se comporte à nouveau comme I . Le nouveau port interagira avec l'émetteur de la requête.

Par exemple, les interfaces d'accès à un compte en banque peut être décrites comme suit : un composant *Banque* propose l'interface serveur *AccesCompte* qui permet à un client de s'authentifier (avec un nom d'utilisateur et un mot de passe), et d'accéder au composant *Compte* lui appartenant. Ce composant propose, quant à lui, une interface *Operations* qui permet de créditer et/ou débiter le compte correspondant. Cependant, cette interface, de part les modalités que nous introduisons, demande que le compte ne reste jamais négatif : *OpsNegatif* oblige le client à créditer le compte jusqu'à ce qu'il soit positif. *OpsNegatif* présente une particularité intéressante : bien qu'estampillé **must ?**, le message *debiter* n'est pas obligatoire car l'interface reviendra dans l'état *OpsNegatif* ; par contre un message *crediter* avec un montant suffisant est bien obligatoire.

AccesCompte = **may ?** [authentication (string, string) ; **must !** [accorde (Operations) ; **0** + refuse ; **0**]]*

Operations = **may ?** [crediter (integer) ; **must !** [solde (integer) ; Operations] + debiter (integer) ; **must !** [solde (integer) ; Operations + solde_negatif (integer) ; OpsNegatif]]

OpsNegatif = **must ?** [crediter (integer) ; **must !** [solde (integer) ; Operations + solde_negatif (integer) ; OpsNegatif] + debiter (integer) ; **must !** [solde_negatif (integer) ; OpsNegatif]]

L'introduction de modalités mène à un modèle sous-jacent qui est une sorte de Système de Transition Etiqueté, dans lequel chaque état est soit **may** soit **must** [LAR 95]. Cela a un impact assez fort sur les règles de compatibilité de la section 3.2.

Le langage d'interface que nous avons défini ci-dessus a quelques limitations. Il est par exemple impossible d'envoyer et de recevoir des messages en même temps ; cepen-

dant, nous proposons la solution qui consiste à instancier deux ports : un qui s'occupera des réceptions, l'autre des envois. Une seconde limitation porte sur le fait que l'on ne peut pas utiliser les deux modalités **may** et **must** sur une même action, par exemple $I = (\mathbf{must} \ ? \ M) + (\mathbf{may} \ ! \ N)$. Mélanger les modalités pose un problème d'équité : dans cet exemple, le composant associé peut ne jamais consommer M parce qu'il sera toujours occupé à envoyer N . Pour empêcher cela, nous proposons d'introduire des contraintes temporelles, imposant par exemple que "**must** ? M " soit honoré dans un délai de 5 unités de temps. La contrainte de temps peut aussi bien être liée à un domaine temporel (comme dans le π -calcul temporisé de [BAI 02]), ou basé sur le nombre de réduction comme dans [KOB 02]. Les travaux futurs sur ce sujet devront essayer de simplifier au maximum la vérification de la compatibilité des interfaces : c'est ce principe qui a imposé ces limitations.

3.2. Règles de compatibilité

Dans cette section, nous définissons le prédicat symétrique $Comp(I, J)$ comme étant : " I et J sont compatibles entre elles". La compatibilité entre les interfaces I et J est définie comme suit (en supposant que si l'une est en envoi, l'autre est en réception) :

$$\begin{aligned} I = \mathbf{must} \ ? \ m &\text{ implique } J = \mathbf{must} \ ! \ m \\ I = \mathbf{may} \ ? \ m &\text{ implique } J = \mathbf{must} \ ! \ m \text{ ou } J = \mathbf{may} \ ! \ m \text{ ou } J = \mathbf{0} \\ I = \mathbf{must} \ ! \ m &\text{ implique } J = \mathbf{must} \ ? \ m \text{ ou } J = \mathbf{may} \ ? \ m \\ I = \mathbf{may} \ ! \ m &\text{ implique } J = \mathbf{may} \ ? \ m \end{aligned}$$

Les règles de compatibilité sont définies à partir de plusieurs relations élémentaires de compatibilité : entre modalités, messages et types. Nous définissons la compatibilité entre les modalités comme la relation booléenne symétrique $Comp_{\text{mod}}(\text{mod}_I \ [! \ ?], \text{mod}_J \ [! \ ?])$. Sa table de vérité est la suivante :

J	I				
	must ?	may ?	must !	may !	0
must ?			✓		
may ?			✓	✓	✓
must !	✓	✓			
may !		✓			
0		✓			✓

Nous définissons aussi $Comp_{\text{msg}}$, une relation de compatibilité sur les types de message. Deux types de message sont compatibles si et seulement si ils ont le même nom et leurs arguments sont égaux syntaxiquement¹. La définition formelle donne :

$$\begin{aligned} Comp_{\text{msg}}(M, M') &\triangleq Comp_{\text{msg}}(M(I_1, \dots, I_n), M'(J_1, \dots, J_m)) \\ &\triangleq M = M' \wedge n = m \wedge \forall i, I_i = J_i \end{aligned}$$

Nous définissons alors la relation de compatibilité $Comp(I, J)$ entre deux interfaces comme étant la compatibilité entre les modalités et les messages, et que les transitions doivent mener à des interfaces compatibles. Ceci est formellement défini récursivement comme suit (avec $\rho \in \{?, !\}$, et où $[*]$ indique que la construction $*$ peut être présente ou non) :

1. Il est possible d'utiliser une relation de sous-typage, que nous donnerons pas ici par manque de place

$$\begin{aligned}
& \text{Comp}(I, J) \triangleq \text{Comp}(J, I) & \text{Comp}(\mathbf{0}, \mathbf{0}) \triangleq \text{true} \\
\text{Comp}(\mathbf{0}, \text{mod}_J \rho_J [\Sigma_l M'_l; J_l] [*]) & \triangleq \text{Comp}_{\text{mod}}(\mathbf{0}, \text{mod}_J \rho_J) \\
\text{Comp}(\text{mod}_I ! [\Sigma_k M_k; I_k], & \triangleq \text{Comp}_{\text{mod}}(\text{mod}_I !, \text{mod}_J ?) \\
\text{mod}_J ? [\Sigma_l M'_l; J_l] [*]) & \\
& \wedge (\forall k, \exists l : \text{Comp}_{\text{msg}}(M_k, M'_l) \wedge \text{Comp}(I_k, J_l)
\end{aligned}$$

Par exemple, les interfaces suivantes sont compatibles respectivement avec les interfaces *AccesCompte* et *Operations* présentées plus haut :

Client = **must !** [authentication (string, string) ; **must ?** [accorde (Operations) ; **0**]
+ refuse ; **0**]]

OpsClient = **must !** [crediter (integer) ; **must ?** [solde (integer) ; **0**]
+ debiter (integer) ; **must ?** [solde (integer) ; **0**]
+ solde_negatif (integer) ; OpsClient]]

Cette définition récursive montre que la compatibilité d'une paire d'interfaces est une fonction booléenne de la compatibilité d'un ensemble fini de paires d'interfaces (les interfaces résultant des transitions et celles référencées dans les messages). Cela signifie que la vérification de la compatibilité est similaire à la vérification de l'équivalence d'automates finis, et par conséquent se termine toujours, et peut être effectuée par les techniques standards avec une complexité quadratique selon le nombre d'interfaces (représentant les différents états des interfaces). De part l'abstraction utilisée dans la définition des interfaces, ce nombre est présumé petit par rapport à la complexité du comportement du composant.

4. Respect de contrat

Le langage d'interface présenté dans la section précédente impose des contraintes sur l'interface distante. Ces contraintes impliquent aussi des contraintes sur les composants. Dans cette section, nous présentons une relation de typage entre les composants et le langage d'interface, pour que le composant respecte un contrat décrit par ce langage.

4.1. Règles pour le typage des interfaces

Les définitions de la section 2 sont étendues à la notion de contrat. Un composant a un ensemble de contrat, un pour chaque port. Nous utilisons la notation :

$u : T$ la référence u a le comportement T , qui est un *type* (selon la section 3).
 (B, \tilde{U}) B a les contrats \tilde{U} , un ensemble de $(u : T)$, tel que à chaque référence (port local ou partenaire) est associé un contrat. L'ajout ou la modification d'une référence est noté $\tilde{U} \leftarrow (u' : T')$, et la suppression $\tilde{U} \setminus u$.

Les règles, exposées dans le tableau 4, sont basées sur celles de la section 2, et où Com est abstrait à partir de la structure d'état. Une explication détaillée des règles est donnée dans [CAR 03], une version préliminaire de cet article. Les règles se terminant par "-ERR" correspondent à une violation du contrat ; le prédicat $\text{Must}(T)$ postule que toute référence u qui est typée **must !** n'est pas suspendue par une référence v qui est typée **may ?**.

Les trois premières règles concernent la création et la suppression d'un port dans le composant. L'attachement de références (BIND et BIND-ERR) stipule que la référence du

Par souci de compréhension, nous notons :

$$M_\Sigma \triangleq [\Sigma_k M_k(\tilde{T}'_k); T_k] \quad M_\Sigma^* \triangleq [\Sigma_k M_k(\tilde{T}'_k); T_k]^* \quad m_k \triangleq M_k(\tilde{v}_k)$$

$$\begin{aligned} \text{CREAT} & \frac{u : T \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow B'(\mathbf{P}[u \multimap \perp], \mathbf{R}', \mathbf{T}')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \rightarrow (B'(\mathbf{P}[u \multimap \perp], \mathbf{R}', \mathbf{T}'), \tilde{U} \Leftarrow u : T)} \text{Must}(\mathbf{T}') \\ \text{REMV} & \frac{u : T \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow B'(\mathbf{P} \setminus u, \mathbf{R}', \mathbf{T}')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \rightarrow (B'(\mathbf{P} \setminus u, \mathbf{R}', \mathbf{T}'), \tilde{U} \setminus u)} (T \equiv \mathbf{0} \text{ ou } T \equiv \mathbf{may} ! M_\Sigma) \wedge \text{Must}(\mathbf{T}') \\ \text{REMV-ERR} & \frac{u : T \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow B'(\mathbf{P} \setminus u, \mathbf{R}', \mathbf{T}')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \rightarrow \text{Error}} T \not\equiv \mathbf{0} \text{ et } T \not\equiv \mathbf{may} ! M_\Sigma \\ \text{BIND} & \frac{u : T \quad u' : T' \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow B'(\mathbf{P}[u \multimap u'], \mathbf{R}, \mathbf{T})}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \rightarrow (B'(\mathbf{P}[u \multimap u'], \mathbf{R}, \mathbf{T}), \tilde{U})} \text{Comp}(T, T') \\ \text{BIND-ERR} & \frac{u : T \quad u' : T' \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow B'(\mathbf{P}[u \multimap u'], \mathbf{R}, \mathbf{T})}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \rightarrow \text{Error}} \neg \text{Comp}(T, T') \\ \text{SEND} & \frac{u : T \equiv \text{mod} ! M_\Sigma \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \xrightarrow{u:u'!m_k} B'(\mathbf{P}, \mathbf{R}', \mathbf{T}')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \xrightarrow{u:u'!m_k} (B'(\mathbf{P}, \mathbf{R}', \mathbf{T}'), \tilde{U} \Leftarrow u : T_k)} \blacktriangleleft \\ \text{SEND-ERR} & \frac{u : T \equiv \text{mod } \rho M_\Sigma[*] \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \xrightarrow{u:u'!m'} B'(\mathbf{P}, \mathbf{R}', \mathbf{T}')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \rightarrow \text{Error}} \neg m' : M_\Sigma \vee \rho = ? \\ \text{RECV} & \frac{u : T \equiv \text{mod} ? M_\Sigma \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \xrightarrow{u:u'?m_k} B'(\mathbf{P}', \mathbf{R}', \mathbf{T}')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \xrightarrow{u:u'?m_k} (B'(\mathbf{P}', \mathbf{R}', \mathbf{T}'), \tilde{U} \Leftarrow u : T_k, \Leftarrow \tilde{v}' : \tilde{T}'_k)} \blacktriangle \\ \text{RECV}^* & \frac{u : T \equiv \text{mod} ? M_\Sigma^* \quad B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \xrightarrow{u:u'?m_k} B'(\mathbf{P}', \mathbf{R}', \mathbf{T}|u'')}{(B(\mathbf{P}, \mathbf{R}, \mathbf{T}), \tilde{U}) \xrightarrow{u:u'?m_k} (B'(\mathbf{P}', \mathbf{R}', \mathbf{T}|u''), \tilde{U} \Leftarrow u'' : T_k, \Leftarrow \tilde{v}' : \tilde{T}'_k)} \blacktriangle \\ \text{MUST-ERR} & \frac{B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow B'(\mathbf{P}', \mathbf{R}', \mathbf{T}')}{B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \rightarrow \text{Error}} \neg \text{Must}(\mathbf{T}') \end{aligned}$$

$$\blacktriangleleft \triangleq \tilde{v}_k : \tilde{T}'_k \wedge (\forall v \in \tilde{v}_k, \text{peer}(v) \Rightarrow (v \notin \text{CoDom}(\mathbf{P}) \wedge v \notin \mathbf{R}')) \wedge \text{Must}(\mathbf{T}') \wedge u' \notin \mathbf{P}$$

$$\blacktriangle \triangleq \text{len}(\tilde{v}') = \text{len}(\tilde{T}'_k) \wedge \text{Must}(\mathbf{T}') \wedge u' \notin \mathbf{P}$$

$$\neg m' : M_\Sigma \triangleq m' = M'(\tilde{v}') \wedge \forall k, M' \neq M_k \vee \neg \tilde{v}_k : \tilde{T}'_k$$

$$\text{Must}(\mathbf{T}') \triangleq \forall u \in \mathbf{T}', (u : \mathbf{must} ! M_\Sigma) \Rightarrow \forall v, u \mapsto^* v, \neg (v : \mathbf{may} ? M_\Sigma)$$

Tableau 4. Règles de compatibilité composant-interface

partenaire doit être compatible avec le port auquel elle est attachée ; le détachement d'une référence partenaire est lui permis à n'importe quel moment.

Les quatre règles suivantes régissent les émissions et réceptions de message. Il est à noter qu'une référence *peer* qui est envoyée ne doit pas être attachée à un partenaire, et doit être retirée de l'ensemble \mathbf{R} ; cela assure la propriété point-à-point de la référence *peer* (soit : cette référence n'est connue que du partenaire). Les règles RECV et RECV* décrivent le comportement normal lors de la réception d'un message – les deux règles diffèrent par la duplication due à la construction *. Nous ne vérifions pas le type des arguments, car le message ayant été envoyé, il l'a été selon le type de l'émetteur ; comme l'émetteur doit être compatible avec le récepteur, nous sommes sûrs que les arguments sont bien typés. Les règles données ici pour l'envoi et la réception correspondent à une interaction externe ; pour les interaction entre les ports d'un même composant, des règles différentes doivent être utilisées : elles réduisent deux étapes de transition (un ! et le ? correspondant) en une seule. Ces règles ne sont pas données ici, par manque de place.

La dernière règle, MUST-ERR, est utilisée pour les cas d'erreurs où une transition mène à un ensemble \mathbf{T} tel que $\text{Must}(\mathbf{T})$ soit fausse.

4.2. Composant honorant un contrat

Un composant respectant un contrat, noté $B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \models \tilde{U}$, est tel que les réductions successives ne mèneront jamais à *Error* :

$$B(\mathbf{P}, \mathbf{R}, \mathbf{T}) \models \tilde{U} \quad \text{ssi} \quad \forall B', \tilde{U}' \text{ tel que } (B, \tilde{U}) \rightarrow^* (B', \tilde{U}') : (B', \tilde{U}') \not\rightarrow \text{Error}$$

4.3. Exemple : application à un composant de gestion de compte

Nous reprenons l'exemple de la section 3 concernant un accès à un compte bancaire. Nous ne détaillons que le composant *Compte*, que nous avons codé en Erlang [ARM 96], et dont nous reproduisons une partie dans le tableau 5. `compte(S)` respecte bien le type *Operations* : sur réception de `crediter`, seul `solde` est retourné, tandis qu'avec `debiter`, soit `solde` soit `solde_negatif` est retourné. La remarque s'applique de même pour `compteneg(S)`.

5. Propriétés garanties par les règles de compatibilité

Jusqu'à présent, nous avons défini les compatibilités entre un composant et le type de ses interfaces, et entre les interfaces. Dans cette section, nous étudions les propriétés d'un assemblage de composants, et prouvons des propriétés de sûreté et de vivacité.

5.1. Assemblage de composants

Nous définissons un assemblage de composants comme une configuration de composants ayant chacun leur contrat, et prêts à interagir via un médium de communication. Cet assemblage a les propriétés :

- chaque référence partenaire désigne un port d'un composant de la configuration,
- les seuls attachements de ports sont les références serveur attachées aux *peers*,

<pre> % Compte avec un solde positif % (implémente le type Operations) compte(S) -> receive {From, crediter, C} -> R = S + C, From ! {solde, R}, compte (R); {From, debiter, D} -> R = S - D, if R < 0 -> From ! {solde_negatif, R}, compteneg (R); true -> From ! {solde, R}, compte (R) end end. </pre>	<pre> % Compte avec un solde négatif compteneg (S) -> receive {From, crediter, C} -> R = S + C, if R < 0 -> From ! {solde_negatif, R}, compteneg (R); true -> From ! {solde, R}, compte (R) end; {From, debiter, D} -> R = S - D, From ! {solde_negatif, R}, compteneg (R) end. </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Tableau 5. Code en Erlang du composant Compte.

– tous les ports sont actifs, sur des tâches indépendantes.

$$\begin{aligned}
 \mathcal{A} &= \{(B_1(\mathbf{P}_1, \mathbf{R}_1, \mathbf{T}_1), \tilde{U}_1), \dots, (B_n(\mathbf{P}_n, \mathbf{R}_n, \mathbf{T}_n), \tilde{U}_n), \text{Com}\} \\
 \text{avec } \begin{cases} \forall i, u : & u \in \mathbf{R}_i \Rightarrow \exists j \text{ tel que } u \in \mathbf{P}_j \\ \forall u, v, i : & (u \multimap v) \in \mathbf{P}_i \Rightarrow \text{peer}(u) \wedge \text{server}(v) \\ \forall u \in \cup \mathbf{P}_i : & \mathbf{T}(u) = \rho^a \end{cases}
 \end{aligned}$$

Dans sa configuration initiale, un assemblage englobe uniquement les liaisons client/serveur. Cependant, alors qu'il évolue, des liaisons point-à-point peuvent apparaître.

Un assemblage sain (ou correct) est un assemblage où chaque composant satisfait les contrats de ses interfaces, et où les ports reliés entre eux ont des interfaces compatibles entre elles.

$$\mathcal{A} \text{ est sain ssi } \begin{cases} \forall i : & B_i \succ \tilde{U}_i, \\ \forall u : T_u, v : T_v, i : & (u \multimap v) \in \mathbf{P}_i \Rightarrow \text{Comp}(T_u, T_v) \end{cases}$$

5.2. Préservation du type et propriété de consommation des messages

La première propriété, P_{sr} , d'un assemblage sain stipule que l'assemblage restera sain tout au long de son évolution. Ce genre de propriété est aussi appelée préservation du type. P_{sr} indique que "une configuration de composants ne mène jamais à *Error*" :

$$P_{sr} \triangleq \forall \mathcal{C} : \mathcal{A} \rightarrow^* \mathcal{C}, \mathcal{C} \not\rightarrow \text{Error}.$$

Théorème 1 (Préservation du Type) Si \mathcal{A} est sain, alors $\mathcal{A} \models P_{sr}$

Preuve 1 La preuve se fait par induction structurelle sur les règles de transition. La propriété est satisfaite en observant que la seule façon qu'une configuration a de mener à *Error* est en violant les règles de compatibilité. \square

Nous définissons aussi P_{mc} : "tous les messages envoyés sont finalement consommés".

$$P_{mc} \triangleq \forall u, v, i, M : (u \multimap v) \in \mathbf{P}_i, (\mathcal{C} \xrightarrow{u:v!M} \mathcal{C}') \Rightarrow \exists \mathcal{C}'', \mathcal{C}''' \text{ tel que } \mathcal{C}' \xrightarrow{*} \mathcal{C}'' \xrightarrow{v:u?M} \mathcal{C}'''$$

Corollaire 1 (Consommation des messages) *Si \mathcal{A} est sain, alors $\mathcal{A} \models P_{mc}$, modulo l'équité.*

Ce corollaire est une conséquence du théorème 1, l'utilisation des files d'attente FIFO et la contrainte qu'un port en réception est actif. Le modulo de l'équité est inévitable, car les règles de consommation de message rentrent en compétition les unes avec les autres.

5.3. Absence d'interblocage externe

L'interblocage externe représente une situation où un ensemble de ports sont bloqués à cause d'un cycle de dépendance. Le cas le plus simple d'interblocage externe est :

$$(u \multimap u') \wedge (v \multimap v') \wedge (u!^s \multimap v?^a) \wedge (v!^s \multimap u'?^a)$$

u est bloqué en envoi par v qui attend que v' effectue son envoi, v' étant bloqué par u' qui attend que u envoie son message. Mais le cas général, plus complexe, est donné par :

$$\begin{aligned} \text{Ext_deadlock}(\mathcal{C}) &\triangleq \exists (t_k)_{1..n} \in \text{tâches}(\mathcal{C}), \exists (u_k)_{1..n}, (v_k)_{1..n} \quad \text{tel que} \\ &\quad t_k = \dots \multimap v_k \multimap \dots \multimap u_k ?^a \\ &\quad \wedge (\forall 1 \leq k < n, (v_{k+1} \multimap u_k)) \wedge (v_1 \multimap u_n) \\ P_{\text{edf}} &\triangleq \forall \mathcal{C}, \mathcal{A} \xrightarrow{*} \mathcal{C} \Rightarrow \neg \text{Ext_deadlock}(\mathcal{C}) \end{aligned}$$

Théorème 2 (Absence d'interblocage externe) *Si \mathcal{A} est sain, alors $\mathcal{A} \models P_{\text{edf}}$*

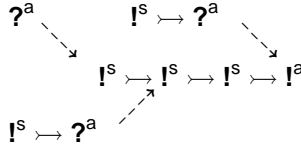
La preuve du théorème 2 est fastidieuse. Même si les interfaces sont mutuellement compatibles, il n'est pas évident qu'un interblocage ne surviendra jamais entre les composants (les ports d'un composant peuvent être suspendus en attente de réception d'un message d'un autre port, ce qui mène à des dépendances potentielles entre les tâches, rajoutées aux dépendances internes entre les ports).

Le problème d'absence d'interblocage a soulevé l'attention récemment. Un travail sur ce sujet très proche du notre a été fait par Naoki Kobayashi [KOB 02], où l'auteur a des actions similaires au **may** et **must** (en termes de capacité et d'obligation), mais la communication est synchrone, et la vérification n'apparaît pas comme étant compositionnelle.

Preuve 2 (Absence d'interblocage externe) L'idée de la démonstration du théorème 2 est la suivante.

Nous définissons une nouvelle relation de dépendance entre les ports, nommée dépendance externe, et notée \dashrightarrow , relatant les communications entre les ports distants. Par exemple : $u?^a \dashrightarrow v!^s$. Nous utilisons ensuite des *arbres de dépendance* pour visualiser les relations de dépendance. Un arbre de dépendance est un arbre orienté dans lequel les nœuds sont de la forme $u\rho^\sigma$, et les arêtes correspondent aux deux relations de dépendance \multimap et \dashrightarrow , dirigées des feuilles vers la racine (les références *idle* ne sont pas prises en compte ; il est évident à partir des règles sur les composant que ces références n'auront

jamais de dépendance). Ainsi, les arbres de dépendance correspondent aux graphes représentant les relations obtenues en fusionnant les deux relations de dépendance. Un exemple d'un tel arbre :



Les arbres de dépendance évoluent avec le comportement des composants. Quelques évolutions sont par exemple la fusion entre deux arbres, le changement d'état d'un nœud...

Nous montrons ensuite par induction structurelle que l'absence de cycle dans les arbres de dépendance est un invariant. Comme par définition cette propriété est satisfaite à l'état initial (\mathcal{A} commence avec un ensemble de tâches indépendantes), l'absence d'interblocage externe est conservée par les dérivations. \square

5.4. Propriété de vivacité, sous condition

L'assemblage de composant a un problème de vivacité : un port peut être éternellement suspendu du fait de la divergence d'un calcul interne ou d'un dialogue sans fin entre deux ports. Il n'est donc pas possible de prouver une propriété de vivacité stipulant "chaque port arrivant à un état **must ?** (ou **must !**) recevra (ou enverra) finalement un message" :

$$P_{\text{must}} \triangleq \forall \mathcal{C}, u, i : \mathcal{A} \rightarrow^* \mathcal{C}, (u : \text{must } \rho M_{\Sigma}) \in \tilde{U}_i \text{ avec } \rho \in \{?, !\} \Rightarrow \\ \exists \mathcal{C}', \mathcal{C}'', v \text{ tel que } \mathcal{C} \rightarrow^* \mathcal{C}' \xrightarrow{u:v\rho M_k} \mathcal{C}''$$

Cependant, nous pensons que cette propriété de vivacité est vérifiée sous les conditions :

- un calcul dans un composant se termine toujours ;
- un port suspendu qui devient actif doit envoyer son message avant de se suspendre ;
- un port qui a une boucle dans son comportement deviendra oisif dans le futur.

De toute façon, ces propriétés ne peuvent être prouvées que si le code source des composants est disponible.

6. Conclusion & travaux futurs

Nous avons présenté un concept de contrat comportemental que nous avons appliqué à un modèle de composant intégrant le multi-tâche, le passage de référence, et les modèles de communication point-à-point et client/serveur. Nos contrats sont utilisés pour la vérification au plus tôt de la compatibilité entre composants, dans le but de garantir des propriétés de sûreté et de vivacité. La compatibilité est décrite formellement dans cet environnement, comme une composition de conformité interne entre les composants et leurs interfaces, et de conformité entre les interfaces.

Dans le contexte d'une conception basée composants, la vérification qu'un composant respecte un contrat par rapport à ses interfaces est en charge du concepteur du composant, qui effectue ce travail une fois pour toutes. La certification du contrat peut être maintenue

par une autorité de certification, pour garantir par exemple à n'importe quel récipiendaire d'un composant mobile ou accessible publiquement que ce dernier n'aura pas d'autre action que celles décrites par ses interfaces.

La vérification de la compatibilité des interfaces devrait être au contraire effectuée au moment où le composant est relié à un autre (soit à l'exécution pour du code mobile, c'est-à-dire lorsque le composant migrant arrive à destination). Nous avons montré que cette vérification peut être faite efficacement à l'aide de techniques standards de vérification d'automates à états finis. La complexité plus élevée de la vérification de la conformité du composant et de ses interfaces est reportée à la compilation, qui peut alors utiliser les techniques de vérification pour les ensembles à états infinis.

Nous avons appliqué notre modèle sur un exemple de compte en banque ; nous devons cependant vérifier la rentabilité de notre approche en pratique, surtout vis-à-vis de l'expressivité du langage d'interface que nous avons proposé (nos travaux futurs s'orientent d'ailleurs vers le langage Erlang). La conformité de notre modèle de composant avec des notations concrètes doit être étudiée : effectuer des modifications sur le modèle de composant peut avoir des conséquences sur les classes de propriétés qui peuvent être garanties. Nous pouvons également observer que les règles de compatibilité peuvent être exprimées en termes de formules de logique temporelle : cela permettrait de prouver un ensemble de propriétés dans un cadre logique plus riche.

Remerciements. Les auteurs de l'ENST ont été partiellement financés par le projet RNTL ACCORD et par le projet IST MIKADO. L'auteur de l'Université de Florence a été partiellement financé par le projet 5% SP4 du Ministère Italien de l'Université et de la Recherche. Le troisième auteur a également été financé par l'ISTI de l'*Italian National Research Council*. Nous remercions également Arnaud Bailly pour ses conseils fructueux.

7. Bibliographie

- [ALF 01] DE ALFARO L., HENZINGER T. A., « Interface Automata », *ESEC/FSE-01*, vol. 26, 5 de *SOFTWARE ENGINEERING NOTES*, ACM Press, 2001.
- [ARM 96] ARMSTRONG J., VIRDING R., WIKSTRÖM C., WILLIAM M., *Concurrent Programming in Erlang*, Prentice Hall, 2nd édition, 1996.
- [BAI 02] BAILLY A., « Assume / Guarantee Contracts for Timed Mobile Objects », PhD thesis, ENST, décembre 2002.
- [CAR 03] CARREZ C., FANTECHI A., NAJM E., « Behavioural Contracts for a Sound Assembly of Components », *Proc. of FORTE 2003*, Sept. 2003.
- [KOB 99] KOBAYASHI N., PIERCE B. C., TURNER D. N., « Linearity and the Pi-Calculus », *ACM Transactions on Programming Languages and Systems*, vol. 21, n° 5, 1999.
- [KOB 02] KOBAYASHI N., « A Type System for Lock-Free Processes », *INFCTRL : Information and Computation (formerly Information and Control)*, vol. 177, 2002.
- [LAR 95] LARSEN K., STEFFEN B., WEISE C., « A Constraint Oriented Proof Methodology Based on Modal Transition Systems », *TACAS'95*, vol. 1019 de *LNCS*, 1995.
- [NAJ 99] NAJM E., NIMOUR A., STEFANI J.-B., « Guaranteeing liveness in an object calculus through behavioral typing », *Proc. of FORTE/PSTV'99*, Oct. 1999.
- [NIE 95] NIERSTRASZ O., « Regular Types for Active Objects », *Object-Oriented Software Composition*, p. 99–121, Prentice-Hall, 1995.