Vérification de conformité des interactions entre composants

(projet COLOSS)

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière
B.P. 92208
44322 NANTES CEDEX 3
prenom.nom@univ-nantes.fr

— Composants, services, architectures de logiciels, vérification —

RAPPORT DE RECHERCHE

N° 04.11

Janvier 2005

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Vérification de conformité des interactions entre composants: (projet COLOSS)

18 p.

Vérification de conformité des interactions entre composants

(projet COLOSS)

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Résumé

Ce rapport de recherche pose les bases d'un cadre formel pour la définition, la composition et la vérification de propriétés des composants logiciels. Après avoir défini le modèle de composants et la composition, nous étudions les moyens d'effectuer rigoureusement des vérifications de conformité des interactions entre composants en nous basant sur les systèmes de transitions qui décrivent les services des composants.

Catégories et descripteurs de sujets : D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

Termes généraux : Composants, Services, Vérification de conformité

Mots-clés additionnels et phrases : Composants, Services, Spécification formelle, Conformité d'interaction

1 Introduction

Le développement de logiciels à partir de composants [10, 7, 4] suscite un certain intérêt mais soulève des questions non encore résolues. Plusieurs paramètres conditionnent la réussite du développement à grande échelle de logiciels à base de composants prédéfinis : la disponibilité de bibliothèque de composants fiables, la disponibilité d'outils de recherche de composants, la disponibilité de langages expressifs de composition des composants et surtout sur la disponibilité d'outils de vérification du bon usage de composants.

Notre cadre de travail se focalise essentiellement le dernier paramètre. L'objectif est de fournir au concepteur d'architectures à base de composants, des moyens de contrôle sur l'usage des composants qu'il souhaite réutiliser. Pour ce faire, nous nous appuyons sur un formalisme simple de modélisation et de composition de composants. L'utilisation d'un composant se fait à travers les services qui constituent son interface. Il est important de respecter les conditions d'utilisation d'un service pour bénéficier des fonctionnalités de celui-ci. Si le respect de la signature d'un service au moment de son appel apparait comme un prérequis facilement vérifiable, le test de conformité a priori, de l'interaction entre un service et son appelant pendant le déroulement du service n'est pas une tâche facile. En effet l'interaction peut être très simple et se résumer en un "appel-réponse" ou alors peut être plus complexe lorsque le déroulement du service nécessite la collaboration de l'appelant. En plus des signatures, la description des comportements des services est necessaires aux appelants. Il est important de détecter, au moment du développement par assemblage des composants, les défauts qui pourraient conduire à un mauvais fonctionnement du système développé. Par exemple un blocage du système global peut surgir en cas de mauvaise interaction entre l'appelant et un service appelé d'un composant. Pour assurer un certain niveau de **correction des composants** et des **assemblages de composants**, l'analyse formelle des descriptions des services par rapport aux propriétés attendues du composant est nécessaire ; des tests de conformité de l'interaction entre les composants doivent aussi être effectués au moment de leur assemblage. On peut ainsi garantir la bonne marche d'un assemblage. Pour cela, la spécification formelle des comportements des composants et de leurs services est nécessaire. On peut ainsi vérifier que les composants qui interagissent ont des **comportements conformes** l'un à l'autre ; c'est-à-dire que les exigences (attente de données, synchronisation) de l'un sont satisfaites par l'autre et vice-versa.

Dans ce travail, nous explorons les moyens d'effectuer rigoureusement ces vérifications de conformité d'interactions en nous appuyant sur un modèle formel de composants. Pratiquement cela permettra de détecter les *incompatibilités de comportement* entre composants (ou services) qui interragissent. Par exemple, un service attend une valeur alors que l'autre ne peut en fournir dans cet état de leur évolution conjointe. L'utilisation d'un modèle formel permet de faire abstraction des détails d'implantation des composants afin d'avoir des techniques générales de raisonnement adaptables facilement à divers environnements spécifiques de mise en œuvre des composants.

Dans la section 2 du rapport, nous donnons un aperçu du modèle formel utilisé dans ce travail. Dans la section 3, nous présentons un formalisme de spécification et de composition associé au modèle. Nous présentons dans la section 4 les grandes lignes de l'analyse de conformité des interactions entre les services des composants. Enfin, nous situons notre modèle vis à vis d'autres propositions dans la section 5. Une petite conclusion termine le document.

2 Un modèle formel pour les composants

L'idée de caractériser les composants par une *interface* constituée de *services offerts* et de *services requis* est aujourd'hui communément admise [1, 7].

Un service offert réalise une fonctionnalité ; Un service requis est la déclaration d'un besoin : si un service s1 offert par un composant c1 a besoin d'appeller un service s2 d'un autre composant de son environnement, on dira que le composant c1 a pour service requis s2. L'environnement du composant est défini lorsque celui-ci se trouve dans un *assemblage* (de composants). De plus, nous considérons que les composants peuvent avoir un espace d'états et des propriétés associées.

2.1 Assemblage de composants

Un *assemblage* est un ensemble de composants qui coopèrent pour la construction d'une application logicielle. Les liaisons entre composants sont les liaisons entre les services requis par les uns et les services offerts par les autres. La figure 1 illustre une configuration pour un système client-serveur.

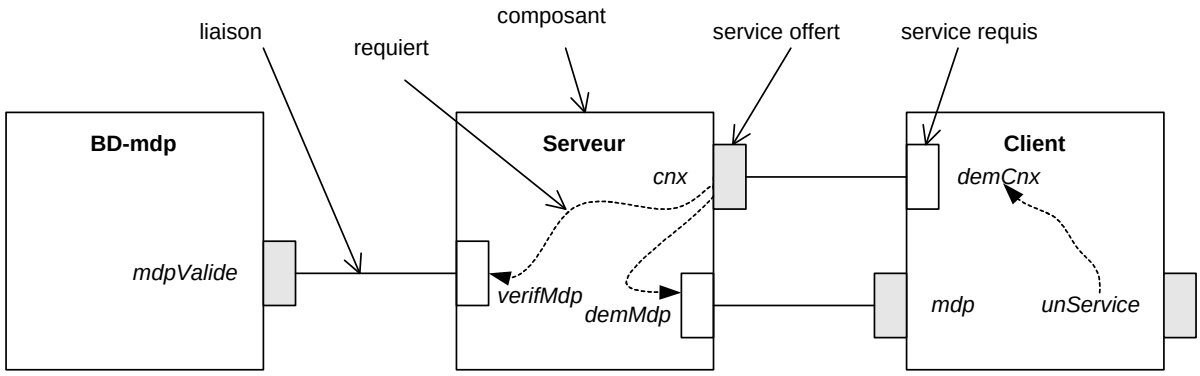

Figure 1 : *Configuration à trois composants*

Pour réaliser un service (offert), un composant peut invoquer des services requis, c'est-à-dire des services offerts par d'autres composants.

Un service offert peut avoir lui-même besoin de services requis. Par exemple, pour réaliser le service de connexion (`cnx`), le composant `Serveur` requiert deux services (*demMdp* et *verifMdp*) pour demander un mot de passer et vérifier un mot de passe. Dans la définition du composant `Serveur` (Figure 1), aucune hypothèse n'est faite sur les composants qui offriront les services requis, un composant qui offrirait les deux services pourrait convenir.

2.2 Spécification d'un composant

Un composant est une entité logicielle utilisable (à travers ses fonctionnalités) pour développer d'autres entités logicielles. On peut distinguer différents niveaux d'abstraction dans la perception du composant ; par exemple un composant peut être exécutable, un composant peut être abstrait (non encore exécutable). Par conséquent un composant peut être la description d'une entité logicielle utilisable par d'autres composants.

Un composant[1] est caractérisé par :

- son nom : il désigne et identifie le composant parmi d'autres composants,
- son état : les informations détenues et les contraintes sur ces informations,
- son interface d'entrée : les services offerts par le composant,
- son interface de sortie : les services requis par le composant,
- son comportement : la description des services et des conditions d'utilisation et d'enchaînement de ces services,
- ses propriétés : les conditions requises pour une utilisation et un fonctionnement correct du composant,
- ...

Formellement, un composant est défini par :

- T un ensemble de types prédéfinis ;
- V un ensemble de variables ;
- V_T un ensemble de variables ayant chacune un type prédéfini ou un type construit à partir des types de base, $V_T \subseteq V \times T$;
- I l'initialisation des variables de V_T ;
- A un ensemble fini d'actions élémentaires ;
- S un ensemble fini de noms de services,

[1] On ne tient pas compte dans cette section des composants formés d'autres composants.

– $\mathcal{D}_S$ la description de chaque service. La spécification précise d'un service est donnée dans la section 2.3. $\mathcal{D}_S$ est partitionné en $\mathcal{D}_{S_o}$ (les services offerts) et $\mathcal{D}_{S_r}$ (les services requis).
– S_{o_v} un ensemble fini de noms de services offerts *visibles* par l'environnement du composant ;
– S_{r_v} un ensemble fini de noms des services requis visibles ;
– $\mathcal{C}_{S_o}$ les contraintes d'interaction entre les services du composant[2].
– $\mathcal{C}_I$ un ensemble de contraintes (invariant) exprimant des propriétés du composant (e.g. de sûreté ou de vivacité).

La notion de visibilité correspond au fait que certains services correspondent à des sous-fonctionnalités d'autres services et ne font pas partie de l'interface d'accès au composant. Nous développons ce point dans les sections qui suivent.

2.3 Services

Un service est défini par une interface et éventuellement un comportement[3]. Ce dernier met en évidence la spécification du déroulement d'un service. Ainsi, nous ne limitons pas la description d'un service à sa *signature* mais nous l'étendons à la *précondition* d'appel du service, à la *postcondition* du déroulement du service et à la description du comportement du service. Toutes ces informations ne sont pas nécessaires pour les services requis. Par défaut, un service requis n'a pas de comportement.

2.3.1 Interface d'un service

L'interface d'un service s comprend la signature, la précondition d'appel et la postcondition du déroulement du service. Le déroulement d'un service peut être l'occasion d'un échange complexe d'informations entre l'appelant et l'appelé qui ne se résume pas au passage de paramètres et au retour de l'appel. Cet échange repose sur des services requis et offerts, donnés dans l'interface sous la forme de deux ensembles :

– s_{o_i} un ensemble fini de noms de services offerts non-visibles (avec $s_{o_i} \cap S_{o_v} = \emptyset$) ;
– s_r un ensemble fini de noms des services requis (e.g. le service *cnx* requiert un service *mdp*).

2.3.2 Comportement d'un service offert

Un service offert est spécifié par un **comportement**, c'est-à-dire des enchaînements d'actions qui peuvent être des actions élémentaires, des invocations de services ou des communications.

– Une action élémentaire est un traitement qui ne nécessite pas d'autres services ou composants ; ce peut être une opération de base (sur des entiers, chaînes de caractères...) ou bien l'invocation d'un service du composant qui lui-même est composé d'actions élémentaires.
– On peut invoquer des services (internes) du composant ou des services requis (externes) au composant. Un service interne peut faire partie de l'interface du composant ou être utilisable uniquement dans le cadre du service spécifié, on le qualifie alors de **sous-service**.
– Une communication est un échange entre deux composants dans le cadre d'un service offert.

2.3.3 Spécification d'un service

Un service est spécifié par un quadruplet $\langle \sigma,\ P,\ Q,\ \mathcal{B} \rangle$ où σ est la signature, P la précondition et Q la postcondition et $\mathcal{B}$ est un système de transitions étiquetées (par des noms de services ou d'actions) définissant le comportement. $\mathcal{B}$ est défini pour les services offerts. Par exemple, le service offert *cnx* du composant `Serveur` (Figure 1) est spécifié comme suit :

```
cnx (nom : String) =
PRE -- précondition P
SPEC
{ init e0 ;   -- e0 est un état initial
```

[2] Cet aspect n'est pas développé dans ce rapport.
[3] *behaviour* en anglais ; pour garder la terminologie utilisée dans le domaine des systèmes interactifs

```
    final e4 ;  -- e4 est un état final
    e0 - !!demMdp() -> e1 ,
                -- invocation du service requis demMdp chez l'appelant
    e1 - demMdp?resMdp(m:String) -> e2 ,
                -- réception du mot de passe sur le service demMdp
    e2 - rep := verifMdp(nom, m) -> e3 ,
                -- invocation du service requis verifMdp
    e3 - !resCnx(rep) -> e4
                -- resultat de la connexion envoyé à l'appelant
  }
  POST -- post condition Q
```

La syntaxe des transitions est `étatSource - varRésultat := [garde] action -> étatCible`. La garde est facultative, c'est une expression logique. Dans l'expression `[rep = ok] ack := true`, les `[]` dénotent une garde.

Un service peut appeler d'autres services du même composant ou d'un composant externe. De ce fait, lorsque nous décrivons un service par un système de transition (automate à états), les étiquettes sur les transitions décrivant le comportement d'un message véhiculent :

- un appel à un autre service (un service ou un sous-service du même composant ou bien un service d'un autre composant). Par exemple, la transition issue de l'état *e2* invoque un service requis. Le type du composant fournisseur du service est connu lorsque le service requis est lié à un service offert d'un autre composant.
- une émission ou une réception. Les étiquettes préfixées par "!" dénotent des émissions vers l'environnement. Les étiquettes préfixées par "?" dénotent des réception de l'environnement. Elles caractérisent les messages de communication. Nous nous sommes inspirés du langage CSP de Hoare. La syntaxe générale d'une primitive de communication est $serviceRequis(!\,|?)\; nomDeService(param*)$.
  Lorsque la variable $serviceRequis$ est précisée, elle dénote un composant qui offre ce service. Lorsqu'elle n'est pas précisée, on communique avec le composant qui a invoqué le service en cours d'exécution. Le type du composant demandeur du service est connu lorsque le service offert est lié à un service requis d'un autre composant.
- un appel à une action élémentaire.

Par souci de simplicité et d'uniformité, nous mimons la notation des communications, en préfixant les appels de services par "!!" et les attentes d'appels par "??". Examinons maintenant les services (`unService` et `mdp`) du côté `client` (Figure 1). On suppose que l'invocation de la demande de connexion apparaît dans le service `unService` du composant `Client`. Ensuite l'interaction est poursuivie avec l'appel de `mdp`.

```
unService () =
PRE true
SPEC
{ init e0 :
  e0 - demCnx(monNom) -> e1 ,    -- invocation du service requis demCnx
  e1 <mdp> ,                       -- le (sous)service mdp est invocable à ce stade
  e1 - demCnx?resCnx(b:Bool) -> e2 -- attente du résultat de la demande de connexion
  ...
}
POST true
```

```
mdp () =
PRE true
SPEC
   { init e0, e1 :
     final e1 :
     e0 - !resMdp(monMdp) -> e1
   }
POST true
```

En résumé, les actions se notent comme suit :
– action élémentaire : `nomAction`.
– invocation de service
  – avec le demandeur : `! !service(param)`.
  – autre : `service(param)`.
– communication
  – avec le demandeur : `!message(param)` en émission, ou `?message(param)` en réception,
  – sur un service requis : `service !message(param)` en émission, ou `service ?message(param)` en réception.

Une représentation graphique facilite la lecture des systèmes de transitions.

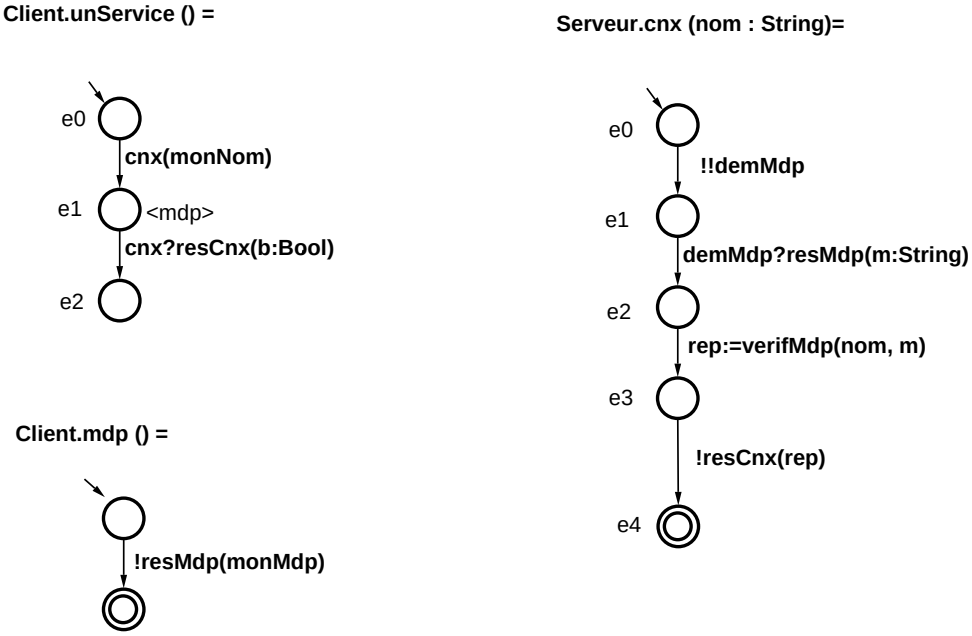

Figure 2 : *Automates des trois services de deux composants*

A chaque état du système étiqueté, on peut associer un ensemble de services offerts, ou de sous-services invocables. La notation `e1 <mdp>` indique que le sous-service *mdp* est invocable dans l'état `e1` ; la suite du traitement est poursuivie dans le sous-service. Pratiquement, cela permet de simplifier la description du système de transitions (graphe orienté) d'un service, en étiquettant certains noeuds par des noms de sous-services x ; la description d'un sous-service est lui même un système de transitions. A partir d'un état `e` du système de transitions d'un service, on peut définir plusieurs sous-graphes que nous noterons $G(e, x)$ par la suite.

3 Un cadre simple de spécification et de composition

Le modèle formel présenté plus haut, nous donne un cadre simple pour spécifier des composants. On utilise quelques clauses pour distinguer les divers constituants du modèle : VARIABLES, TYPES, INITIALISATION, etc.

Pour l'assemblage des composants, nous introduisons un opérateur nommé compose, qui permet de construire un nouveau composant en assemblant un ou plusieurs composants.

L'opérateur compose a pour paramètres :
– un ensemble non vide de composants,
– la description explicite des *liens* entre les services (offerts et requis) des différents composants,
– une définition de son interface. Dans cette interface, on peut mettre des services de ses composés, éventuellement redéfinis (par exemple, en en changeant l'interface/le typage) et de nouveaux services.
– de nouveaux services éventuels.

Un lien entre un service requis sr et un service offert so est noté simplement sous la forme d'un couple (sr, so). Le lien explicite quel service (offert) de quel composant est utilisé pour satisfaire un service requis d'un autre composant. Le composant résultat de la composition liste ses propres services offerts et requis, ses signatures, etc.
Nous offrons la possibilité de redéfinir des interfaces de service sous la forme :
`[serv1 :ancien_profil1/nouv_profil1; serv2 :ancien_profil2/nouv_profil2; ...]`.

Illustration

Nous définissons les deux composants `BD-mdp` et `Serveur` de la figure 1 comme ci-après.

```
Composant BD-mdp
interface
  offerts : {mdpValide, ...}
  requis : {}
variables
  bd : String x String ;  ...
initialisation
  bd := empty ;
  ...
services
  res : Bool <- mdpValide(nom,
                    mdp : String) =
  PRE   true
  SPEC
     {res := (bd(nom) = mdp)
     }
  POST  res : Bool
end
```

```
Composant Serveur
interface
  offerts : {cnx, ...}
  requis  : {verifMdp, demMdp, ...}
variables       ...
initialisation  ...
services
  cnx (nom : String) =
  PRE  true
  SPEC
  {init e0 :
   final e4 :
   e0 - !!demMdp() -> e1 ,
   e1 - demMdp?resMdp(m:String) -> e2,
   e2 - rep := verifMdp(nom, m) -> e3,
   e3 - !resCnx(rep) -> e4 }
  POST  ...
end
```

Nous définissons maintenant en utilisant l'opérateur compose, un composant `Serveur complet`, par assemblage des composants `BD-mdp` et `Serveur`.

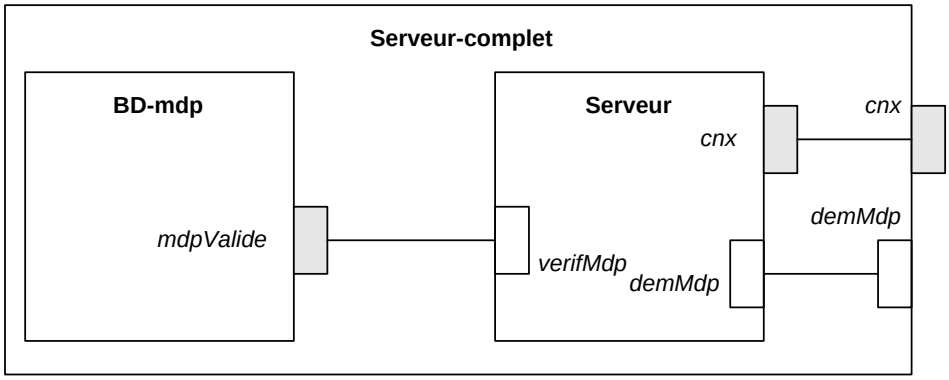

Figure 3 : *Composition de deux composants*

```
Composant Serveur-Complet
   compose({BD-mdp, Serveur},
      /* les liens */
      {(mdpValide, verifMdp)}
               )
interface
    offerts : {cnx}
    requis  : {demMdp}
```

```
services
    ...
```

4 Analyse des composants et assemblages

Les composants et leur assemblage peuvent être analysés sous diverses facettes. Ici nous nous préoccupons uniquement de l'analyse de la conformité des interactions entre services appelants et appelés de façon à détecter les incompatibilités. On suppose que deux composants ou plus sont impliqués dans une composition. La plupart des travaux sur les interactions [5, 11, 3] utilisent des variantes du modèle des automates à états [5, 11]. Ils testent la compatibilité entre les automates représentant des services ; l'explosion combinatoire est la principale difficulté rencontrée. Dans notre proposition nous apportons une solution satisfaisante en considérant un comportement indépendamment de son envionnement.

4.1 Analyse des interfaces

Sans détailler ce point, on vérifie les règles de typage et les pré- et post-conditions. Une manière d'implanter la vérification consiste à traduire les interfaces dans un langage tel que Z, B ou des spécifications algébriques.

4.2 Analyse de composants interagissants

Dans le système de transitions décrivant le comportement d'un service, nous avons de façon générale, des appels de services ou des communications (échanges synchrones de message de type émission/réception). Dans le cas des appels de service, il y a un retour d'appel. L'analyse peut se résumer à la vérification de conformité des interfaces (signatures identiques) entre appelant et appelé. Dans le cas des communications, en plus des appels/retours de services, il y a des communications synchrones (à la manière des produits d'automate ou des algèbres de processus) sur des actions élémentaires.

On a une *incompatibilité de comportements* (ou encore comportements non conformes) lorsque l'appelant d'un service (supposé correctement décrit), n'est pas en mesure de satisfaire les exigences du service appelé au moment de son déroulement. Par conséquent, un *comportement compatible* avec un service est un deuxième comportement qui peut interagir convenablement avec le premier. Lorsqu'il n'y a pas d'interaction, le problème de compatibilité ne se pose pas ; par exemple des actions élémentaires ou des appels de sous-services (sans interaction) font transiter d'un état à un autre. Nous montrons qu'un composant et ses services peuvent être décrits et surtout analysés indépendamment du contexte dans lequel ils seront utilisés. En effet, on ne peut savoir à l'avance le contexte dans lequel le composant sera utilisé.

4.2.1 Analyse de compatibilité des comportements

Nous traitons les services qui interagissent lorsque des composants sont composés. Les *messages en entrée* d'un état s (notés $input_m(s)$) d'un comportement de service sont les messages venant de l'environnement et qui sont reçus dans cet état. Les *messages en entrée* du comportement $\mathcal{B}$ (notés $Input_m(\mathcal{B})$) d'un service sont définis par l'union des $input_m(s)$ des états de $\mathcal{B}$.
Les *messages en sortie* d'un état s (notés $output_m(s)$) d'un comportement de service sont les messages émis de cet état à destination de l'environnement. Les *messages en sortie* du comportement $\mathcal{B}$ (notés $Output_m(\mathcal{B})$) d'un service sont définis par l'union des $output_m(s)$ des états de $\mathcal{B}$.
Les *appels de service* d'un état s (notés $call_m(s)$) d'un comportement de service sont les appels de services d'autres composants ou des appels de sous-services du composant courant. Les *appels de service* du comportement $\mathcal{B}$ (notés $Call_m(\mathcal{B})$) sont en conséquence définis par l'union des messages $call_m(s)$ des états s de $\mathcal{B}$. De la même façon nous définissons les attentes d'appel $waitCall_m(s)$) et $WaitCall_m(\mathcal{B})$. En effet dans un état donné, l'appel d'un service ou d'un autre est attendu sinon il ne peut être traité.

Pour la correspondance entre messages en sortie d'un service et les messages en entrée du service impliqué dans une interaction, nous utilisons l'hypothèse de la communication synchrone. Les services utilisent dans ce cas un alphabet commun.

Définition 4.1 *Collaboration requise.*
Soit un service $se = \langle \sigma, P, Q, \mathcal{B} \rangle$ avec son comportement $\mathcal{B} = \langle S, L, \delta \rangle$; on suppose que l'environnement avec lequel s'effectue l'interaction utilise un ensemble d'états S' et s'appuie aussi sur une modélisation à base de système de transitions.

La collaboration requise pour interagir avec $\mathcal{B}$ est tout comportement $\mathcal{B}'$ symétrique à $\mathcal{B}$ au sens où : i) pour chaque état s, depuis l'état initial de $\mathcal{B}$, les messages en entrée de s sont les messages en sortie de s' qui est l'état courant du comportement de l'environnement, ii) les messages de sortie de s sont les messages d'entrée de s' et iii) les messages d'appels correspondent aux attentes d'appel. De plus, les appels d'un service externe peuvent correspondre au déclenchement et au déroulement (avec interaction possible) d'un sous-système de transitions dans l'autre comportement (comme si on passait d'un état initial à un état final pour un sous-système de transitions adhoc du cté du service appelé).

$$\forall s \in S. \exists s' \in S'. \quad input_m(s) = output_m(s') \ \wedge \ output_m(s) = input_m(s') \ \wedge \ call_m(s) = waitCall_m(s')$$

Tout service ayant un comportement similaire à la *collaboration requise* peut interargir avec le service. Il a un comportement conforme (noté $\overline{\mathcal{B}}$) dans la suite.

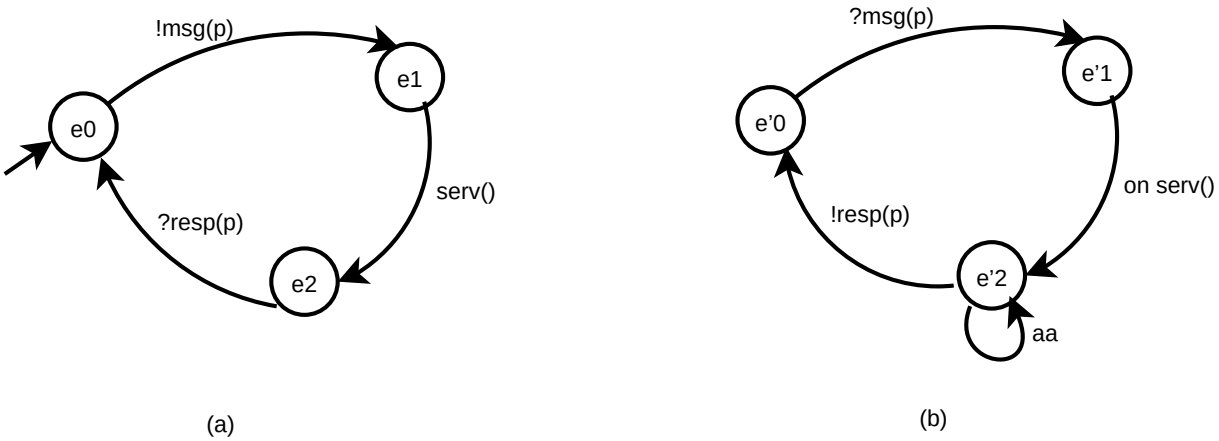

Figure 4 : *Spécification d'un service et une collaboration requise*

La figure 4(b), donne un exemple de collaboration requise pour le service de la figure 4(a).

Définition 4.2 *Comportement strictement conforme (Strict Matching Service).*
Un service se' a un comportement strictement conforme pour un service se si le comportement $\mathcal{B}'$ de se' est equivalent (il s'agit ici de l'équivalence de systèmes de transitions [2]) à la collaboration requise pour le comportement de se. On note $\mathcal{B}' \equiv \overline{\mathcal{B}}$

Définition 4.3 *Comportement non strictement conforme (Non-strict Matching Service).*
Un service se' a un comportement ($\mathcal{B}'$) non strictement conforme à un service se lorsque la collaboration requise de se est un sous-ensemble du comportement proposé par se'. On note $\mathcal{B}' \sim \overline{\mathcal{B}}$

$$\forall s \in S. \exists s' \in S'. \quad input_m(s) \subseteq output_m(s') \ \wedge \ output_m(s) \subseteq input_m(s') \ \wedge \ call_m(s) \subseteq waitCall_m(s')$$

Ces définitions constituent le fil rouge pour la mise au point des algorithmes de test de conformité des interactions. Ces algorithmes sont nécessaires lors de l'assemblage de composants.

4.2.2 Problèmes annexes

Signalons, dans cette section, le problème de l'évaluation des gardes pour tester si des traces d'évaluation sont possibles ou pas.

5 Travaux connexes

Dans un modèle à composant, on trouve habituellement des composants et des relations (ou connexions) entre composants. Il s'agit là des seuls éléments communs à tous les modèles à composants. En effet, comme pour le paradigme objet, il existe de nombreux modèles de composants ; certains étant même une conversion d'anciens modèles aux standards des composants. Pour le reste les définitions des composants et de leurs relations varient d'un modèle à l'autre.

Le composant est l'unité de base d'une architecture logicielle. Par rapport à un objet il est un grain de définition "plus gros", plus autonôme et encapsulé, avec une symétrie retrouvée entre interfaces d'entrées et de sorties, l'existence de plusieurs interfaces (bornes) [6]. L'approche par composant relève donc d'une modélisation à grande échelle avec configuration et reconfiguration des modules (les composants). Habituellement, la description des interfaces des objets (par exemple CORBA/IDL) prend en compte la description de variables et de services offerts mais pas de requis. De plus, la spécification de services ne prévoit pas d'interactions, elle est basée sur le typage des invocations de services.

Les relations ou connexions entre composants sont de nature très variées. Pour s'en convaincre, il suffit d'observer la taxonomie des connexteurs de Mehta et al. [8]. Les connexions peuvent être des liens statiques ou dynamiques entre composants. On distingue généralement les points de connexion (les ports, les interfaces, les points d'accès...) et les connecteurs (liens entre les points de connexion). Dans notre modèle, les points de connexion sont les services et les connecteurs sont les liens entre services.

Pour l'assemblage des conmposants, on distingue différentes architectures (couches logicielles, tubage, diffusion, frameworks, patterns, repositories... [9] Actuellement, dans notre modèle, l'assemblage se fait par une relation de composition.

6 Conclusion

Nous avons présenté un cadre pour le développement par composants. L'interaction entre services est abordée ici. Notre approche qui traite un service de façon indépendante, limite le problème de l'explosion combinatoire. Le cadre est élaboré avec un souci de généralité et de simplicité. Ce travail est en cours ; nous poursuivons la formalisation, la validation sur des études de cas et des expérimentations avec des plate-formes industrielles.

Références

[1] R. ALLEN et D. GARLAN. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, July 1997.

[2] André ARNOLD. *Systèmes de transitions finis et sémantique des processus communiquants*. Collection Etudes et recherches en informatique. Masson, 1992. ISBN 2-225-82746-X.

[3] P. C. ATTIE et D. H. LORENZ. Establishing Behavioral Compatibility of Software Components without State Explosion. Rapport technique NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, 2003.

[4] K. BERGNER, A. RAUSCH, M. SIHLING, A. VILBIG, et M. BROY. A Formal Model for Componentware. In G. T. LEAVENS et M. SITARAMAN, réds., *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, New York, NY, 2000.

[5] L. de ALFARO et T. A. HENZINGER. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[6] Jean-Pierre Briot FRÉDÉRIC PESCHANSKI, THOMAS MEURISSE. Les composants logiciels : Evolution technologique ou nouveau paradigme ? In *Actes de la Conférence Objets, Composants, Modèles (OCM'2000)*, pages 53–65, Nantes, France, May 2000. In french.

[7] N. MEDVIDOVIC et R. N. TAYLOR. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, january 2000.

[8] Nikunj R. MEHTA, Nenad MEDVIDOVIC, et Sandeep PHADKE. Towards a taxonomy of software connectors. In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM Press, 2000.

[9] M. SHAW et D. GARLAN. *Software Architecture : Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[10] Clemens SZYPERSKI. *Component Software : Beyond Object-Oriented Programming*. AddisonWesley Publishing Company, 1997.

[11] D.M. YELLIN et R.E. STROM. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2) :292–333, 1997.

Table des matières

LABORATOIRE D'INFORMATIQUE DE NANTES-ATLANTIQUE

Vérification de conformité des interactions entre composants (projet COLOSS)

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Résumé

Ce rapport de recherche pose les bases d'un cadre formel pour la définition, la composition et la vérification de propriétés des composants logiciels. Après avoir défini le modèle de composants et la composition, nous étudions les moyens d'effectuer rigoureusement des vérifications de conformité des interactions entre composants en nous basant sur les systèmes de transitions qui décrivent les services des composants.

Catégories et descripteurs de sujets : D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

Termes généraux : Composants, Services, Vérification de conformité

Mots-clés additionnels et phrases : Composants, Services, Spécification formelle, Conformité d'interaction

LINA, Université de Nantes
2, rue de la Houssinière
B.P. 92208 — F-44322 NANTES CEDEX 3