Spécification d'architectures en Kmelia: hiérarchie de connexion et composition

Pascal André — Gilles Ardourel — Christian Attiogbé

LINA - FRE CNRS 2729
2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France
(Pascal.Andre,Gilles.Ardourel,Christian.Attiogbe)@univ-nantes.fr

RÉSUMÉ. *Dans cet article, nous présentons la spécification d'architectures logicielles dans* Kmelia*, un modèle à composants basé sur les services.* Kmelia *permet de définir un modèle simple d'architecture fondé sur la description des composants, des services et de leurs interactions, le raisonnement sur des modèles incomplets et le raffinement. Nous traitons plus particulièrement de i) la hiérarchisation des connexions par une description fine des protocoles et des interfaces de services, ii) la composition de services et de composants supportant la hiérarchisation et iii) la vérification outillée de composabilité des éléments de l'architecture qui permet de détecter statiquement des incompatibilités dans les assemblages (visibilités, assertions, comportement dynamique).*

ABSTRACT. *In this article, we describe the specification of software architectures with* Kmelia*, a component model based on services. Using* Kmelia *one can define a simple model of architecture that supports the description of the components and their interactions, the reasonning on incomplete models and refinement. More precisely we study the following issues: i) the hierarchisation of connections by a fine and flexible approach for the description of protocols; ii) the composition of services and components that support this hierarchisation; iii) assisted verification of the composability of the architecture elements that permit to detect statically the assembly mismatches (visibility, assertions, dynamic behaviour).*

MOTS-CLÉS : *Architecture Logicielle, Composants, Services, Hiérarchisation, Méthodes formelles*

KEYWORDS: *Software Architecture, Components, Services, Hierarchisation, Formal Methods*

1. Introduction

Nous nous plaçons à un niveau abstrait, dans lequel l'architecture est une description abstraite et modulaire du système. A ce niveau, l'architecture est perçue comme une collection de composants (au sens d'entitités logicielles), une collection de connecteurs (pour décrire les interactions entre composants) et des configurations, c'est-à-dire des assemblages de composants et de connecteurs [ALL 97]. L'accent est mis sur la modularité, qui est un critère important pour la réutilisabilité, la fiabilité et l'évolutivité du logiciel. On utilise des ADLs (*Architecture Description Language* pour spécifier ces architectures. Les ADL sont nombreux et variés [CLE 96, MED 00, BAR 05], de même que les types de connecteurs [MEH 00]. Les modèles couvrent tout ou partie des besoins en termes de langage, de sémantiques et d'outils. Dans [MED 00] les auteurs relèvent des insuffisances pour la spécification des propriétés non-fonctionnelles des systèmes, un manque de fondement sémantique pour l'expression de contraintes et du raffinement (composant, connecteur et configuration) et un manque d'outils pour la reconfiguration et l'évolution. Nous situons notre travail dans la seconde problématique, et plus précisément sur la formalisation des architectures permettant la vérification de propriétés et le raffinement.

Nous avons proposé un modèle simple, formel et outillé de description d'architecture logicielle dans lequel on puisse à la fois concevoir simplement les architectures et disposer de conditions d'assemblage riches et flexibles. Ce modèle est nommé Kmelia comme le langage de spécification qui l'accompagne [ATT 06]. L'idée est d'enrichir suffisamment les interfaces des éléments architecturaux pour déterminer si leur assemblage dans une architecture possède de bonnes propriétés (correction, fiabilité, sûreté...) ou pas. Dans Kmelia les éléments architecturaux sont des composants (au sens *Component Based System Engineering* ou CBSE [LAU 06, CRN 02]) dans lesquels les services sont des entités de première classe. Cette vision nous permet, contrairement à la plupart des autres approches de type CBSE, de nous rapprocher des architectures orientées services (*Service Oriented Architecture* ou SOA [ERL 05, PAP 03]), notamment pour la composition. Nous considérons le développement de composants indépendants de toute plateforme d'implantation (*composants abstraits*), qui interagissent via des services. L'assemblage des composants dépend ainsi directement des liaisons entre services.

Dans cet article, nous nous penchons plus particulièrement sur les problèmes de structuration hiérarchique, de flexibilité et de formalisation dans les spécifications d'architectures. La hiérarchisation se base essentiellement sur les relations de composition (de services, de liens et de composants) ; elle permet de masquer la complexité des éléments de l'architecture (lisibilité, traçabilité), sert de base au processus de conception et de raffinement (transformations d'architectures). La flexibilité des descriptions améliore la réutilisabilité des composants et des services (sous-services optionnels, utilisation partielle de composants, renommage, interfaces raffinables). La formalisation permet la vérification de propriétés des architectures et la détection statique des incompatibilités dans les assemblages (visibilités, assertions, comportement dynamique). Les principales contributions de ce travail sont : *i)* la hiérarchisation des

connexions par une description fine des protocoles et des interfaces de services, *ii)* la composition de services et de composants incluant la hiérarchisation et *iii)* la vérification outillée de composabilité des éléments de l'architecture qui permet de détecter statiquement des incompatibilités dans les assemblages (signatures, visibilités, assertions, comportements dynamiques).

La suite de l'article est organisée de la façon suivante : la section 2 présente les principales caractéristiques de notre modèle, du point de vue de la hiérarchisation dans les architectures. Dans la section 3 nous traitons plus spécifiquement de la composition de services. La section 4 résume les contributions en termes d'outils et de vérification autour de Kmelia. La section 5 situe notre approche parmi des travaux similaires. Enfin nous évaluons le travail et indiquons des perspectives dans la section 6.

2. Hiérarchisation dans une architecture à base de composants et de services

Dans cette section, nous présentons la spécification d'architectures logicielles en Kmelia, un modèle abstrait et un langage à composants. Kmelia permet de modéliser des architectures logicielles supportant la hiérarchisation des composants, des services et des liens pour une bonne lisibilité, flexibilité et traçabilité dans la conception d'architectures. Nous illustrons cette présentation par l'exemple classique du guichet automatique bancaire (GAB), qui a pour avantage de ne pas nécessiter une longue description informelle pour sa compréhension.

2.1. *Eléments d'une architecture logicielle*

Une architecture logicielle est définie par une collection de composants (unités de mémorisation d'informations et de calcul), une collection de connecteurs (pour décrire les interactions entre composants) et une ou plusieurs configurations, c'est-à-dire des assemblages de composants et de connecteurs. Ces composants encapsulent des services dans des interfaces, où on distingue les *services offerts*, qui réalisent une fonctionnalité, des *services requis*, qui déclarent les besoins des fonctionnalités, conformément à [ALL 97, MED 00]. Dans ce contexte, nous considérons des *composants abstraits*, indépendants de leur environnement et par conséquent non exécutables directement. Dans Kmelia, les composants sont assemblés sur leurs services par des *liens d'assemblage* (les *connecteurs*) dans des *assemblages* (les *confi gurations*). Kmelia se différencie d'autres modèles d'architectures par un style architectural épuré, dans lequel les connecteurs sont simplement des liaisons et non des entités de première classe, et par le fait que les services ne sont pas de simples opérations mais des entités de première classe avec des interfaces spécifiques. Dans un assemblage, un service joue aussi un rôle de *port virtuel* (appelé *canal* dans Kmelia) sur lequel circulent des messages. Les communications sont synchrones. Nous préciserons par la suite ces concepts lors de leur utilisation. Pour une définition formelle, consulter [ATT 06]. Noter qu'un composant décrit un état (ensemble de variables typées), un invariant d'état et différentes contraintes (sous forme de prédicats).

2.2. *Hiérarchisation des liens d'assemblage*

Étudions l'architecture *simplifi ée* d'un guichet automatique bancaire GAB, qui propose deux services à ses utilisateurs : le `retrait` d'argent et la `consultation` de compte. Dans ces deux cas, le GAB a besoin de la carte et du code de l'usager. La figure 1 représente une architecture pour un tel GAB. Cette architecture est abstraite en ce sens qu'elle est indépendante de tout mécanisme d'implantation. Cette première architecture met en évidence quatre composants et en particulier le composant central `BASE_GAB` qui propose quatre services bancaires. On s'intéresse essentiellement aux relations entre le composant principal `BASE_GAB` et l'interface client `INTERFACE_CLI`. Le composant `INTERFACE_CLI` offre un service `requete`, qui modélise un accès par le client. Il est présenté en partie dans l'annexe A. On supposera que ce service de requête peut invoquer le service `retrait` du composant `BASE_GAB` durant son fonctionnement.

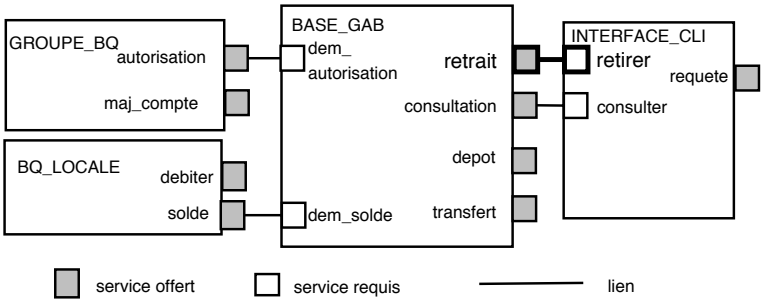

Figure 1 : *Assemblage abstrait et simplifi é du Guichet Automatique Bancaire*

Les composants sont indépendants de leur contexte d'assemblage. Dans la définition du composant `BASE_GAB`, aucune hypothèse n'est faite sur les composants qui offriront les services requis, un composant qui offrirait les deux services conviendrait (si par exemple une banque locale délivre les autorisations). De même, aucune hypothèse n'est faite sur les "clients" des services offerts (le service `consultation` pourrait être requis par un autre composant). Un *lien* d'assemblage est la satisfaction d'un service requis par un service offert. Par exemple, le service `requete` offert par le composant `INTERFACE_CLI` a besoin (par hypothèse) d'appeler un service `retirer`, ce dernier est *lié* dans l'assemblage au service `retrait` offert par le composant `BASE_GAB`.

L'assemblage proposé est relativement abstrait en ce sens qu'il masque des détails de spécification. Ce gros grain de spécification est utile pour avoir une vue d'ensemble du système et pour envisager réutilisation de composants existant. Une architecture plus concrète peut être obtenue par raffinement de composants ou de services. Intéressons-nous au cas où l'architecture concrète diffère significativement de l'architecture abstraite au niveau de la granularité des services. Par exemple l'architecture de la figure 2 décrit plus finement le lien `retrait-retirer`. Elle a pu être obtenue pour les raisons suivantes. D'une part, un composant concret `BASE_GAB` existe et rend

globalement le service `retrait` même s'il nécessite l'appel de plusieurs services. D'autre part, la communication nécessaire pour la réalisation du service `retrait` est suffisamment complexe pour nécessiter un découpage qui factorise plusieurs "petits" services utilisables par d'autres services ou composants. Cette architecture « plus concrète » est conforme à l'architecture abstraite, du point de vue de l'ensemble des composants et des services rendus. Cependant, la différence de granularité dans l'utilisation des services ne permet pas d'associer simplement les liens en gras qui participent au retrait dans l'architecture concrète avec le lien `retrait-retirer` de l'architecture abstraite.

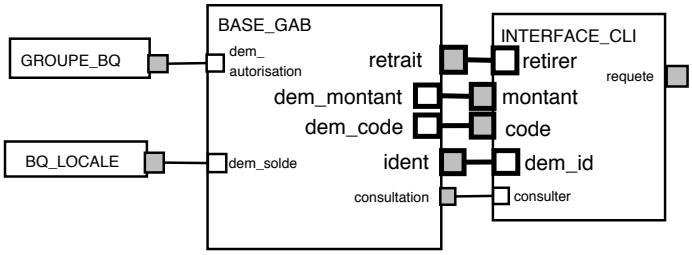

Figure 2 : *Un assemblage plus concret du GAB, à granularité de service plus fi ne*

Afin d'assurer la traçabilité entre ces deux modèles, il faut que la notion de service global `retrait` englobant toute la communication maintenant distribuée entre `retrait`, `identification`, `code` et `montant`, soit présente dans le modèle concret. En d'autres termes, il faut qu'il apparaisse que le service global `retrait` est une combinaison des services `retrait`, `ident`, `code` et `montant`. On peut les combiner dans un `composant-service` `RETRAIT`, comme le montre la figure 3.

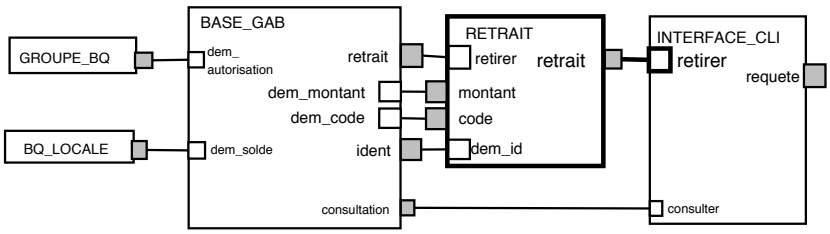

Figure 3 : *Un assemblage plus concret du GAB, avec un patron de conception*

La figure 3 illustre l'introduction de la notion de service global `retrait` dans l'architecture en utilisant le seul concept de composant comme unité de structuration (s'il n'y a pas de composition de liens/services mais juste une granularité composant). Une partie du composant `INTERFACE_CLI` est déplacée dans un composant `RETRAIT` pour réintroduire le lien `retrait-retirer`. On aurait pu également le faire de façon symétrique pour la partie du composant issue de `BASE_GAB`. Dans cette solution, la

traçabilité est obtenue grâce à l'utilisation d'une sorte de *patron de conception architectural* et donc au prix d'une d'une complication de la structure qui rend plus difficile la lisibilité et qui risque de provoquer une confusion entre service et composant.

Pour remédier à ce problème, nous proposons une solution basée sur la notion de composition de liens et de services. La composabilité des composants permettant de considérer une architecture à plusieurs niveaux de détails (on peut y naviguer selon les niveaux "hiérarchiques"), il nous a paru intéressant de considérer que des services et des connexions soient eux aussi composables. La figure 4 illustre une solution dans laquelle les services `retrait` et `retirer` ainsi que les liens qui les connectent sont obtenus par composition hiérarchique.

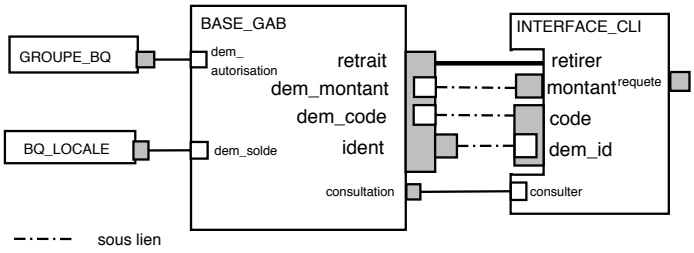

Figure 4 : *Un assemblage plus concret du GAB, par composition de services*

La présence d'un service au sein d'un autre service signifie qu'il est connecté au sein d'un même canal établi par la connexion au service principal. Par exemple, dans la figure 4 la présence du service offert `code` dans le service requis `retirer` établit que le service `code` est appelable par le service qui satisfait `retirer`. Comme on peut le voir sur cette figure, il n'y a pas de niveau maximum de composition. Cependant, il est évident qu'un niveau trop élevé nuit à la compréhension et est un signe de mauvaise modularité ; ce point est abordé en fin de section 2.5. Le passage de l'architecture concrète à l'architecture abstraite est une question de niveau de détail (ou de zoom si on dispose d'une navigation interactive). Bien que cette solution ne s'applique pas à tous les raffinements d'architectures, nous pensons qu'elle est plus naturelle et lisible qu'un patron de conception architecturale. Au problème de flexibilité et de réutilisation, nous préférons une approche langage plutôt qu'une approche motifs de conception, réservant autant que possible ces derniers à la conception du système.

En résumé, la hiérarchisation des liens d'assemblage, qui va de pair avec celle des services, facilite la visualisation et le masquage des connexions par zoom sur les liens d'assemblage. Elle améliore la flexibilité de description des services qui deviennent hiérarchiques avec des sous-services optionnels. Elle permet différentes formes de raffinement sur les services et les composants. Enfin elle correspond à l'idée que ce qui motive la réutilisation d'un composant, ce sont les services qu'il propose et non des points d'accès structurels.

2.3. *Hiérarchisation des interfaces de services en Kmelia*

A la structuration des liens correspond nécessairement une structuration des services. La définition des interfaces de services en Kmelia doit être suffisamment précise pour permettre l'établissement des liens et sous-liens décrits précédemment. La spécification des services et de leurs interfaces est abordée en détails dans la section 3.1, nous n'en considérons ici que l'aspect hiérarchique, appelée la *dépendance de services*. La dépendance de service d'un service `sv` est un quadruplet d'ensembles finis et disjoints de noms de services utilisés dans le cadre du service sv : `subprovides`, `calrequires`, `extrequires`, `intrequires`. Un service listé dans `subprovides` est rendu accessible au travers de la connexion au service `sv` jugé plus global. Ce mécanisme permet une gestion fine de l'encapsulation des services. Un service listé dans `calrequires` doit être fourni par le composant qui est connecté au service global `sv`. Cette contrainte aurait pu être exprimée à l'aide d'un autre mécanisme mais découle naturellement de la formation des sous-liens et s'intègre naturellement à l'interface d'un service. Un service listé dans `extrequires` doit être fourni par un composant qui est connecté au service listé (et donc dans l'interface du composant dans lequel `sv` est défini). Noter qu'un service qui apparaît dans la dépendance d'un autre service mais pas dans l'interface du composant est appelé *sous-service* dans Kmelia. Un service listé dans `intrequires` doit être fourni par de composant qui définit `sv`. Prenons l'exemple du service `retrait` de la figure 4. Pour sa réalisation il requiert deux services de son appelant (`dem_code` pour demander le code de la carte et `dem_montant` pour demander le montant du retrait), il requiert un service d'un composant quelconque (`dem_autorisation` pour vérifier l'autorisation de retrait), et il propose un service `ident`. Les dépendances sont explicitées par des flèches dans la figure 5. La syntaxe Kmelia est la suivante :

```
BASE_GAB
provided retrait (cb : CB)
Interface
   subprovides : {ident}
   calrequires : {dem_code,
                  dem_montant}
   extrequires : {dem_autorisation}
```

```
INTERFACE_CLI
provided requete ()
Interface
   extrequires : {retirer,
                  consulter}
provided code () : Integer
Interface
   calrequires : {dem_id}
required retirer (carte : CB)
Interface
   subprovides : {code}
```

Ces dépendances vont au delà du simple rattachement de services à d'autres qui est nécessaire au regroupement des liens. Elles permettent l'expression de contraintes riches en termes de dépendances et flexibles en termes de structuration. Ainsi, il faut noter sur la figure 4 qu'il n'y a pas de symétrie parfaite entre le service requis `retirer` et le service offert `retrait`. C'est une illustration de la flexibilité offerte par le mécanisme de sous-liens, et surtout du principe de modularité : aucun des deux services concernés ne peut présumer des relations entre les services qu'ils requiert de la part

du composant de l'autre. Le service `retrait` doit être invoqué par un composant qui possède le service `code`, mais ni sa connexion ni son comportement ne dépendent du fait que `code` soit directement accessible ou seulement offert dans l'interface de `retirer`.

2.4. *Composition de composants en Kmelia*

La composition de composants en Kmelia est définie par l'encapsulation d'assemblage de composants (une configuration) dans un composant, qu'on nommera le composé pour plus de clarté. Les services du composé sont ceux qu'il définit (ses services propres) et ceux de ses composants qu'il décide de promouvoir. L'exemple d'architecture de la figure 5 illustre à la fois un assemblage et une composition pour le GAB. Le composant `GAB_SYSTEME` est assemblé avec deux composants représentant l'environnement bancaire, les composants `GROUPE_BQ` et `BQ_LOCALE`. Le nommage des composants dans l'assemblage de la figure 5 signifie qu'on distingue les composants-types de leurs instances dans Kmelia. La structuration est hiérarchique puisque le composant `GAB_SYSTEME` est lui-même composé d'un composant central `BASE_GAB` et d'une interface IHM `INTERFACE_CLI`.

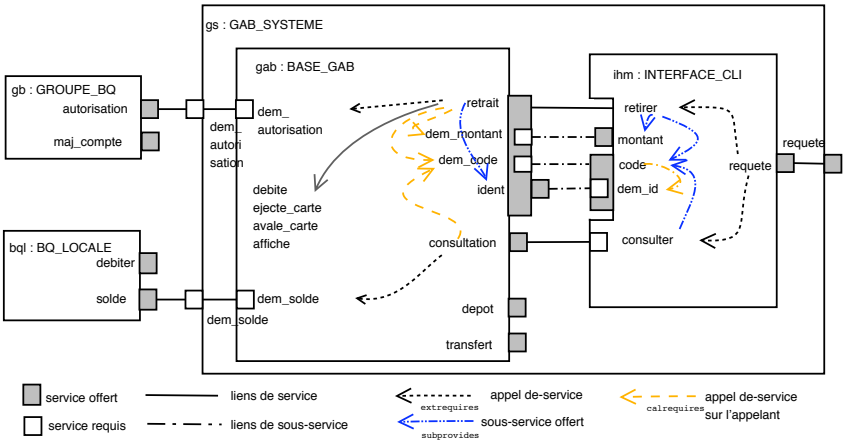

Figure 5 : *Assemblage et composition du guichet automatique bancaire*

Appliquée récursivement, la composition des composants décrit une hiérarchie de composants. A chaque niveau, l'encapsulation induit le masquage des composants d'un composite et de leurs services. Pour être accessible, un service d'un composant doit être *promu* dans le composite par un *lien de promotion*. L'interface d'un service obtenu par promotion peut différer de celle du service d'origine à la condition qu'elle reste compatible avec cette dernière (les conditions de compatibilité sont détaillées dans [ATT 06]). Pour plus de *flexibilité*, nous permettons l'assemblage partiel de composants et le renommage des composants et des services dans un assemblage. Dans un

assemblage, il n'est pas nécessaire de lier tous les services offerts et tous les services requis, pourvu que ces services ne soient pas invoqués. Ainsi, on peut disposer d'un composant riche et complexe mais l'utiliser partiellement pour ne pas développer et tester un nouveau composant. Le renommage des composants et des services permet de les adapter au contexte d'une application en les rendant plus lisibles.

2.5. *Hiérarchisation et méthodologie*

D'un point de vue méthodologique, la composition de composants s'utilise pour affiner la description d'un composant abstrait en un assemblage de composants (*démarche descendante*) ou bien pour structurer des composites (*démarche ascendante*). La composition de la figure 5 illustre la démarche ascendante : le composant `gs` est défini par composition des composants `gab` et `ihm`. Dans une démarche descendante et sur ce même exemple, on raffinera les fonctions internes `ejecte_carte`, `avale_carte`, `debite` et `affiche` du composant `gab` soit par de nouveaux services requis, soit par des services de composants internes (e.g. écran, imprimante, lecteur de carte, distributeur). Composition et raffinement sont intimement liés du point de vue méthodologique. Par ailleurs les choix adoptés pour la hiérarchisation des liens d'assemblage (section 2.2) sont influencés par la hiérarchisation par composition de composants et les liens de promotion qui en découlent.

3. Spécification et composition de services

La hiérarchisation de services se base sur des opérateurs de composition (inclusion obligatoire ou facultative de services) et de délégation (services requis de différentes sources, services internes) qui autorisent une conception ascendante ou descendante. La hiérarchisation des services est décrite de façon abstraite (uniquement des noms) dans leur interface et va de pair avec la hiérarchisation des liens et sous-liens. Dans cette section, nous détaillons la spécification des services et nous présentons les éléments qui permettent d'assurer leur composition et leur flexibilité.

3.1. *Spécification d'un 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. De ce fait, 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 (dynamique) du service. Il est donc possible de décrire [1]) les services à quatre niveaux de précision : signature (nom, paramètres, types), assertions (pre et postconditions), signature

1. et donc de vérifier : aux niveaux de précision correspondent des niveaux de vérification de correction des spécifications de services.

enrichie (relative à la hiérarchisation) et comportement dynamique (automate). Il y a inclusion (hiérarchique) des niveaux de précision et Kmelia autorise une flexibilité dans les descriptions : la spécification peut se limiter à un niveau donné. Par exemple, on peut ne donner que la signature et les assertions d'un service. Cette *flexibilité* favorise *l'interopérabilité* du modèle Kmelia avec d'autres modèles dont les interfaces ne sont pas aussi riches. Par exemple, on peut introduire dans un assemblage des composants munis d'interfaces IDL et vérifier la compatibilité des services en se restreignant au niveau signature. Cette flexibilité apparaît aussi au niveau des composants et des assemblages. Par exemple, le composant `INTERFACE_CLI` peut fonctionner avec un service de retrait quel que soit l'ordre dans lequel lui sont demandés les montants et le code de la carte, et il est piloté par le service de retrait en ce qui concerne le nombre de ces demandes.

L'**interface d'un service** d'un composant est spécifiée par une signature (le profil du service avec son nom, ses paramètres et le résultat), la précondition, la postcondition, un ensemble de déclarations de variables locales au service et la dépendance de service définie dans la section 2.3.

Le **comportement d'un service** s est un système de transitions étiquetées étendu (ou *eLTS, extended Labeled Transition System*) spécifié par un sexuplet comprenant l'ensemble des états du service, l'ensemble des étiquettes, un état initial, un ensemble non vide des états finaux (un service se termine toujours), une relation de transitions étiquetées entre états et une fonction d'annotation d'états. Cette dernière étend le LTS en associant des sous-services offerts de `subprovides` aux états. L'annotation est à la base de la composition de services. La spécification 1 illustre les définitions précédentes pour le service `retrait`.

Listing 1 – Spécification Kmelia du service `retrait`

```
provided retrait (cb : CB) : Boolean
Interface
   subprovides : {ident}
   calrequires : {dem_code, dem_montant}
   extrequires : {dem_autorisation}
Pre    caisse >= limite
Variables # locales au service
  nbe    : Integer,       # nombre d'essais
  c      : Integer,       # code fourni
  m      : Integer,       # montant fourni
  rep    : Boolean,       # reponse de la demande d'autorisation
  succes : Boolean        # resultat du retrait
Behavior
init i
final f
{ i --{nbe:= 3 ; succes := false } --> e0,
                 # trois essais au plus
  e0 -- __CALLER!!dem_code() --> e1,
     # appel du service requis dem_code de l'apelant
  e1 <ident>, # ident est un sous service invocable ici
```

```
  e1 ---{ __CALLER??dem_code(c) ; # recoit le code
          nbe := nbe -1           # un essai en moins
  } --> e2i ,
  e2i -- __CALLER!rdv() --> e2 ,
  e2 --[c=cb.code] rep :=
       _dem_autorisation!!dem_autorisation(cb.id ,c) --> e3 ,
     # appel du service requis dem_autorisation
     # sur le canal dem_autorisation
  e2 --[c<>cb.code && nbe>0]
          affiche("Relecture du code") --> e0 ,
  e2 --[c<>cb.code && nbe=0]
       {affiche("carte avalee"); avale_carte()} --> e4 ,
     # le retrait echoue , la carte est éconserve
  e3 --[rep] affiche("Lecture du montant") --> e5 ,
     # le groupement accepte la transaction ,
     # le montant est demande
  e3 --[not rep] {affiche("Transaction refusee") ;
            ejecte_carte() } --> e4 ,
     # the groupement refuse la transaction ,
     # le retrait echoue ,
  e4 -- __CALLER!!retrait(succes) --> f ,
     # le service se termine : l'appelant
     # est informe du resultat sur son canal
  e5 -- __CALLER!!dem_montant() -->e6 ,
     # invoque le service dem_montant de l'appelant
  e6 -- __CALLER??dem_montant(m) --> e7 ,
     #  communication reception : attend le montant
  e7 --[m<= cb.limit]
       {debite(c,m) ; ejecte_carte()} --> e8 ,
  e7 --[m > cb.limit]
       affiche("depassement du solde") --> e3 ,
  e8 -- succes := true --> e4
}
Post  caisse <= _pre(caisse) # le montant diminue
end
```

La syntaxe simplifiée des transitions est :

`étatSource -- [ garde ] bloc-action --> étatCible.`

L'étiquette d'une transition est un bloc d'actions optionnellement gardées. Un bloc d'actions est délimité syntaxiquement par les symboles { et } sur la transition. Le résultat d'une action peut être affecté à une variable.
Une action prend l'une des formes décrites ci-après :

– une *action interne* : c'est un traitement qui ne nécessite pas d'invocation de services ; cela peut être un calcul sur un type de base, un calcul fonctionnel ou bien une composition d'actions internes. Une action interne peut, par raffinement du composant, être définie par un service. Ce raffinement des actions constitue un des éléments de la conception descendante des composants.

– une *communication* : c'est un échange synchrone (émission notée par `!` ou réception notée par `?`) de message sur un canal de service. Cette notation est inspirée du langage CSP de Hoare.

– une *invocation* de service : c'est un appel (noté par `!!`) ou un retour (noté par `??`) de service. Trois cas sont distingués : un service interne (offert par le composant lui-même), un service requis auprès de l'appelant, un service requis (son composant fournisseur sera connu à l'assemblage des composants).

La syntaxe des invocations et des communications est :

`canal(!|?|??|!!) sélecteur(paramètres)`.

Le sélecteur est un nom de service pour une invocation et un nom de message pour une communication. La variable `canal` est soit le nom d'un service requis, soit le canal spécifique `__CALLER` qui correspond à l'appelant du service dans le contexte d'un déroulement de service, soit le canal spécifique `__SELF` pour un service interne.

3.2. *Composition de services*

Le comportement (dynamique) des services comprend une fonction d'annotation d'états par des services. Syntaxiquement, la notation `e1i <ident>` indique que le sous-service `ident` est invocable dans l'état `e1` du service `retrait`. En cas d'invocation du service `ident` lorsque le service `retrait` est dans l'état `e1i`, la suite du traitement est poursuivie dans le service `ident`, mais au sein du canal établi par l'appel de `retrait`. Cette notion de sous-services offerts attachés à un état d'automate apporte à la fois de la *flexibilité* dans le modèle (invocation optionnelle de ces sous-services lorsque le déroulement atteint l'état) et un moyen de structuration des automates (hiérarchisation, partage). Selon que la conception est ascendante ou descendante, elle est la cause ou la conséquence de l'apparition de la directive `subprovides` dans le service et d'un sous-lien dans un lien associé à `retrait`. Ce mécanisme constitue une partie évolutive du support pour la composition flexible de services dans Kmelia. L'autre partie de ce mécanisme permet de décrire l'obligation d'appeler un service au sein du dialogue avec un autre service jugé plus global en associant ce sous-service à une transition de l'automate du service le plus global. Il permet la description de contraintes de synchronisation ou d'exclusion entre services et la définition de *protocoles* d'utilisation des services du composant. Ces contraintes sont comparables aux *Component Behavior Protocol* de [PLA 02, GIA 99, PAV 05] à ceci près que notre modèle autorise plusieurs protocoles via les services.

La composition de services dans Kmelia se décline donc selon deux axes.

1) La composition *horizontale* définit une relation de dépendance (utilisation) entre services. Elle est basée sur les appels de service et la bonne interaction entre deux services. Elle permet de définir la composition de composants.

2) La composition *verticale* définit une relation de structuration hiérarchique (inclusion) qui permet de définir de nouveaux services à partir de services existants. Parmi les opérateurs de composition, on trouve les sous-services optionnels rattachés

aux états, les sous-services rattachés aux transitions, etc.

La disponibilité de mécanismes de composition de services facilite la définition de nouvelles abstractions de services sans passer nécessairement par l'introduction artificielle de nouveaux composants.

4. Analyse formelle et outillage

Nous abordons dans cette section les possibilités d'étude offertes par la disponibilité d'un modèle formel. Nous ne détaillons pas les différentes possibilités dans cet article mais nous abordons nos réalisations pour la propriété de composabilité des composants dans un assemblage.

Le modèle Kmelia nous a servi de base de raisonnement pour l'analyse de la composabilité sur des compositions. La composabilité se résume à l'interopérabilité statique et l'interopérabilité dynamique.
La vérification de l'interopérabilité statique se fait à la compilation de la spécification Kmelia par analyse des interfaces en profondeur (sans toutefois tenir compte des assertions) : vérification des signatures de tous les services en jeu, concordance des liens et sous-liens, complétude des services requis pour tout service offert dans la composition (seuls sont traités les services effectivement utilisés dans la composition).
Pour l'interopérabilité dynamique, on explore l'évolution dynamique des échanges. L'interaction entre des composants se traduit par l'interaction de leurs services. Ainsi nous nous basons sur une analyse de l'interaction paire à paire de services (appelant et appelé sur une liaison). L'interopérabilité dynamique est alors assurée lorsqu'il n'y a pas de blocage dans les interactions entre les services considérés, interprétés comme des processus. Sur la base de cette similarité, nous avons formalisé la correspondance entre les services des composants et les processus puis, nous avons développé deux outils de traduction de spécifications Kmelia dans des langages adaptés et outillés MEC [AND 06b] et LOTOS/CADP [ATT 06]. Dans chacun des cas, nous avons détecté des erreurs dans les interactions (erreurs de type, incohérences d'interfaces, problèmes d'ordre dans les messages, problèmes de blocage, problèmes de complétude). L'intérêt de MEC est sa proximité avec les LTS, la traduction est plus simple et l'interprétation des résultats est immédiate. Malheureusement, il ne permet pas le traitement des gardes et des données. A l'inverse LOTOS permet un traitement des gardes et des données (les types sont limités pour éviter l'explosion combinatoire) mais la traduction du LTS d'un service en LOTOS génère un ensemble de processus dont le cardinal est fonction de la complexité et de la taille du eLTS de départ. De fait la traduction n'est pas directe et l'interprétation des résultats est plus délicate.

En complément de l'interopérabilité statique et du contrôle de type, nous devons vérifier la validité des assertions. Dans un premier temps nous avons choisi une approche plutôt traditionnelle : il s'agit de vérifier que l'invariant d'état du composant est préservé lors de l'exécution des services en respectant les préconditions. La véri-

fication effective se fait dans un langage adapté et outillé. Des expérimentations sont en cours avec le langage B [ABR 96].

Nous avons développé un prototype en Java (nommé COSTO : *Component Study Toolbox*)) qui supporte Kmelia et implante les idées développées dans cet article. Dans sa version actuelle, il comprend un compilateur, une interface graphique et des outils pour la visualisation de modèles et la vérification de propriétés. Le compilateur (basé sur ANTLR) permet de lire des spécifications Kmelia et de stocker les spécifications dans une structure abstraite sous forme d'objets. Il réalise une partie des vérifications statiques concernant l'interopérabilité statique. La structure abstraite sert à la vérification de propriétés des modèles et la transformation de modèles. Concernant la vérification de propriétés, des algorithmes complètent la vérification de l'interopérabilité statique et des outils de traduction vers Lotos et MEC parcourent la structure abstraite et génèrent des spécifications dans les langages correspondants. La vérification de l'interopérabilité dynamique est traitée avec les outils de ces langages. Le traitement des assertions n'est pas encore implanté.

5. Discussions

On distingue principalement deux catégories d'approches pour les architectures logicielles [BAR 05] : celles inspirées du développement de logiciels à base de composants (*CBSE*) et celles orientées service (*SOA*). Dans le premier cas [SZY 97, BER 00, ALL 97] l'accent est mis sur la structure statique du système : les éléments logiciels sont des composants assemblés par des connecteurs dans des configurations. Dans le second cas [ERL 05, PAP 03] l'accent est mis sur la structure fonctionnelle du système : les éléments logiciels sont des fonctionnalités (les services) liés par des relations de type collaboration ou composition. L'approche choisie dans Kmelia peut être qualifiée de mixte en ce sens qu'elle comprend des composants et des services : l'*élément architectural* de base est le *service* tandis que le *composant* Kmelia encapsule ses services dans une interface. L'originalité de notre modèle par rapport à la plupart des approches à composants et services [PLA 02, GIA 99, PAV 05, BEY 05] est que les services ne sont pas de simples procédures (ou fonctions) mais des entités composables et combinables. Un service est composé d'autres services (sous-services) et se trouve en interaction avec d'autres services (qu'il requiert ou qui le requièrent). Cette hiérarchisation des services induit une hiérarchisation des liens d'assemblage et fournit des règles d'assemblage non triviales (pas réduites à des correspondances de noms). Kmelia permet la composition d'architectures en encapsulant des assemblages dans des composants, les liens de promotion assurant la connexion des services des composants à ceux des composites et vice-versa.

Comparés à ceux des langages de description d'architectures [ALL 97, MED 00, MEH 00] les connecteurs de Kmelia se résument à des liens, autrement dit des connecteurs qui font simplement une correspondance de noms. En fait un lien sert aussi de canal de communication dans les échanges entre services. Il n'y a pas de comportement spécifique associés à ces liens, qui ne sont pas non plus des entités logicielles

spécifiables. Néanmoins il peuvent avoir différentes sémantiques selon leur cardinalité. Dans une sémantique *monadique* un lien relie deux services au plus. C'est le cas dans la version actuelle du langage Kmelia. Avec la sémantique *polyadique*, les liens peuvent relier plusieurs services par exemple, un service offert peut être lié à plusieurs services requis pour de la diffusion (par exemple dans un forum de discussion), un service requis peut invoquer plusieurs services offerts (choix déterministe ou non du fournisseur). Il faut alors déterminer plus finement le routage des messages (émetteur et receveur ne sont plus implicites). C'est le cas lorsque les connecteurs représentent des ports (canaux) structurels « à la CSP » comme dans [ALL 97, PLA 02, GIA 99].

Pour la description dynamique, nos travaux se rapprochent de ceux traitant la spécification des composants ou services à l'aide d'automates, d'expressions régulières ou d'algèbres de processus avec la vérification de la compatibilité comportementale [ALF 01, VAN 01, ATT 03a, PAV 05, PLA 02, MEH 00]. Notre approche se distingue essentiellement de ces travaux par le fait que les comportements sont associés aux services et non aux composants, ce qui permet une approche bien plus fine des interactions et de la composabilité. Elle se distingue aussi par la souplesse dans l'expression des interactions. La composabilité est basée en partie sur l'interaction entre les comportements. Nous avons élaboré une solution qui fonctionne localement avec les services pris deux à deux. Cette solution limite le problème de l'explosion combinatoire qui est la principale limitation de la plupart des travaux sur les interactions [ALF 01, YEL 97]. Ces derniers utilisent des variantes du modèle des automates à états et leur composition. Dans [ATT 03a, ATT 03b] les auteurs traitent de la correction des compositions de composants avec limitation de l'explosion combinatoire ; dans leur approche plusieurs automates sont utilisés pour un même composant : un automate pour chaque interaction du composant avec un autre composant.

6. Conclusion et perspectives

Nous avons présenté les propriétés de la spécification d'architecture logicielles en Kmelia. En considérant les services comme des entités de base éventuellement complexes et décomposables, Kmelia permet de modéliser des architectures logicielles supportant la hiérarchisation des composants, des services et des liens. Cette particularité évite la confusion entre service et composant qui nuit à la lisibilité et à la tracabilité et autorise l'expression de contraintes d'assemblage dans les interfaces de services. Une description formelle de Kmelia est détaillée dans [ATT 06].

Kmelia offre la possibilité d'exprimer finement des interfaces riches permettant d'étudier et de vérifier a priori la composabilité de composants. Nous avons axé nos traitements sur des concepts éprouvés (systèmes de transitions) disposant des outils aussi bien théoriques que pratiques pour asseoir l'analyse formelle. Cela nous permet la réutilisation d'outils tels que CADP ou MEC. L'environnement logiciel COSTO accompagne notre travail pour des fins d'expérimentations : il intègre un compilateur, une interface graphique et des outils pour la visualisation de modèles et la vérification de propriétés comme la composabilité.

Les perspectives de ce travail concernent le langage, le prototype et leurs applications. Nous travaillons actuellement sur les propriétés de correction des propriétés fonctionnelles des services des composants (expressions et assertions). Parallèlement nous continuons le développement de notre prototype afin de le rendre disponible en *open source* et le tester sur des exemples significatifs. Nous nous intéressons également au points de vue de la méthodologie et du processus de spécification. Par exemple, un trop grand nombre de niveaux de sous-services dans une spécification est le signe d'un problème qui doit être traité par l'utilisation d'opérateurs de réingénierie ou l'introduction d'adapteurs, comme abordé dans [AND 06a].

7. Bibliographie

[ABR 96] ABRIAL J.-R., *The B-Book Assigning Programs to Meanings*, Cambridge University Press, 1996, ISBN 0-521-49619-5.

[ALF 01] DE ALFARO L., HENZINGER T. A., « Interface Automata », *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, ACM Press, 2001, p. 109-120.

[ALL 97] ALLEN R., GARLAN D., « A Formal Basis for Architectural Connection », *ACM Transactions on Software Engineering and Methodology*, vol. 6, n° 3, 1997, p. 213–249.

[AND 06a] ANDRÉ P., ARDOUREL G., ATTIOGBÉ C., « Coordination and Adaptation for Hierarchical Components and Services », *Third International ECOOP Workshop on Coordination and Adaptation Techniquesfor Software Entities (WCAT'06)*, 2006, to appear.

[AND 06b] ANDRÉ P., ARDOUREL G., ATTIOGBÉ C., « Vérifi cation d'assemblage de composants logiciels Expérimentations avec MEC », GOURGAND M., RIANE F., Eds., *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006*, Rabat, Maroc, avril 2006, Lavoisier, p. 497–506.

[ATT 03a] ATTIE P., LORENZ D. H., « Correctness of Model-based Component Composition without State Explosion », *ECOOP 2003 Workshop on Correctness of Model-based Software Composition*, 2003.

[ATT 03b] ATTIE P. C., LORENZ D. H., « Establishing Behavioral Compatibility of Software Components without State Explosion », rapport n° NU-CCIS-03-02, 2003, College of Computer and Information Science, Northeastern University.

[ATT 06] ATTIOGBÉ C., ANDRÉ P., ARDOUREL G., « Checking Component Composability », *5th International Symposium on Software Composition, SC'06*, vol. 4089 de *LNCS*, Springer, 2006.

[BAR 05] BARAIS O., « Construire et maîtriser l'évolution d'une architecture logicielle à base de composants », PhD thesis, LIFL, Université Lille 1, novembre 2005.

[BER 00] BERGNER K., RAUSCH A., SIHLING M., VILBIG A., BROY M., « A Formal Model for Componentware », LEAVENS G. T., SITARAMAN M., Eds., *Foundations of Component-Based Systems*, p. 189–210, Cambridge University Press, New York, NY, 2000.

[BEY 05] BEYER D., CHAKRABARTI A., HENZINGER T. A., « Web service interfaces », *WWW '05 : Proceedings of the 14th international conference on World Wide Web*, New York, NY, USA, 2005, ACM Press, p. 148–159.

[CLE 96] CLEMENTS P. C., « A Survey of Architecture Description Languages », *IWSSD '96 : Proceedings of the 8th International Workshop on Software Specification and Design*, Washington, DC, USA, 1996, IEEE Computer Society, page 16.

[CRN 02] CRNKOVIC I., LARSSON M., Eds., *Building Reliable Component-Based Software Systems*, Artech House publisher, 2002.

[ERL 05] ERL T., Ed., *Service-Oriented Architecture - Concepts, Technology, and Design*, Prentice Hall, Boston, MA, USA, 2005.

[GIA 99] GIANNAKOPOULOU D., KRAMER J., CHEUNG S.-C., « Behaviour Analysis of Distributed Systems Using the Tracta Approach. », *ASE*, vol. 6, n° 1, 1999, p. 7-35.

[LAU 06] LAU K.-K., WANG Z., « A Survey of Software Component Models », rapport n° CSPP-38, mai 2006, School of Computer Science, The University of Manchester.

[MED 00] MEDVIDOVIC N., TAYLOR R. N., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, n° 1, 2000, p. 70–93.

[MEH 00] MEHTA N. R., MEDVIDOVIC N., PHADKE S., « Towards a taxonomy of software connectors », *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, ACM Press, 2000, p. 178–187.

[PAP 03] PAPAZOGLOU M. P., « Service-Oriented Computing : Concepts, Characteristics and Directions », *WISE*, IEEE Computer Society, 2003, p. 3–12.

[PAV 05] PAVEL S., NOYE J., POIZAT P., ROYER J.-C., « Java Implementation of a Component Model with Explicit Symbolic Protocols », *4th International Symposium on Software Composition, SC'05*, vol. 3628 de *LNCS*, Springer, 2005.

[PLA 02] PLASIL F., VISNOVSKY S., « Behavior protocols for Software Components », 2002, IEEE Transactions on SW Engineering, 28 (9), 2002.

[SZY 97] SZYPERSKI C., *Component Software : Beyond Object-Oriented Programming*, AddisonWesley Publishing Company, 1997.

[VAN 01] VANDERPERREN W., WYDAEGHE B., « Towards a New Component Composition Process », *ECBS*, IEEE Computer Society, 2001, p. 322–328.

[YEL 97] YELLIN D., STROM R., « Protocol Specifications and Component Adaptors », *ACM Transactions on Programming Languages and Systems*, vol. 19, n° 2, 1997, p. 292–333.

A. Extrait d'une spécification Kmelia de composant

Listing 2 – Spécification partielle en Kmelia du composant INTERFACE_CLI

```
COMPONENT INTERFACE_CLI
INTERFACE
   provides : {requete}
   requires : {retirer, consulter}
TYPES
   CB   :  {code:Integer, id:Integer, limit:Integer}
VARIABLES
   maCarte : CB,
   monCode : Integer
SERVICES
```

```
#---------- services offerts ----------
provided requete ()
     #Service principal decrivant le comportement du client
Interface
   extrequires : {retirer, consulter}
Variables # locales au service
  b : Integer ,
  res:Boolean
Behavior       # decrit une boucle infinie
init i    final f
{ i --{affiche("Bonjour, inserer votre carte");
        lire(maCarte)} --> e0,
  i -- stop() --> f,
  e0 -- _retirer!!retirer(maCarte) --> e1,
            # invocation du service retrait
  e0 -- _consulter!!consulter(maCarte) --> e10,
            # invocation du service consulter
  e1 <retirer.code>,
     # code est un sous service invocable
     # dans le contexte de 'retirer'
  e1 -- _retirer?rdv() --> e2, # pour le fun
  e2 -- _retirer??retirer(res) --> i,
            # attente du resultat du service retirer
  e10 <consulter.code>,
     # code est un sous service invocable
     # dans le contexte de 'consulter'
  e10 -- _consulter??consulter(b) --> i
     #---le comportement est non specifie ici
}
end
provided code () : Integer      # rend le code de l'utilisateur
Interface
   calrequires : {dem_id}
Variables
  id : String       # id du GAB
Behavior
init e0     final f
{ e0 --{affiche("Entrer votre code");lire(monCode)} --> e1,
  e1 -- __CALLER!!dem_id() --> e2, #__CALLER
  e2 -- __CALLER??dem_id(id) --> e3,
  e3 -- memoriser(id, today) --> e4,
  e4 -- __CALLER!!code(monCode) --> f # renvoie le code
}
end
provided montant () : Integer
...
end
#---------- services requis ----------
required retirer (carte : CB) : Boolean
Interface
```

```
   subprovides : {code}
end
required consulter (carte : CB) : Integer
Interface
   subprovides : {code}
end
END_SERVICES
```