Kmelia : un modèle abstrait et formel pour la description et la composition de compo-sants et de services

Pascal Andre, Gilles Ardourel, Christian Attiogbé

Kmelia : un modèle abstrait et formel pour la description et la composition de composants et de services

Pascal André — Gilles Ardourel — Christian Attiogbé

LINA - UMR CNRS 6241 / F-44322 Nantes Cedex 3, France
Prenom.Nom@univ-nantes.fr

RÉSUMÉ. *Kmelia est un langage et un modèle à composants multi-services où les composants sont abstraits et formels de façon à pouvoir y exprimer des propriétés et à les vérifier. Dans Kmelia un service peut interagir avec son appelant ; il peut encapsuler d'autres services auxquels il donne accès et aussi requérir d'autres services de son appelant ou non. Les services de Kmelia peuvent être paramétrés par des données et sont dotés d'assertions (sous la forme de pré/post-conditions opérant sur les données). Dans cet article nous présentons les principales caractéristiques de Kmelia à travers les moyens de composition de services et de composants qui sont offerts. La composition des composants et des services détermine les possibilités d'interaction ; nous présentons ainsi les différents cas d'interaction entre les services qui sont la base de la composition et des interactions. Nous présentons les méthodes d'analyse formelle élaborées en même temps que l'approche Kmelia et l'outil COSTO que nous développons. Nous illustrons l'article par l'étude de cas CoCoME consacrée à la gestion d'un site de vente de produits à distance.*

ABSTRACT. *Kmelia is both a language and a multi-services component-based model. The Kmelia components are abstract and formal to permit the description and the verification of properties. Within Kmelia a service may interact with its caller ; it can encapsulate other services to which it gives access and it can also require services from its caller or from other components. The Kmelia services can be parameterised with data and they are equipped with assertions which are expressed as pre-post-/conditions operating on the data. In this article we introduce the main features of the Kmelia approach through the provided means for service composition and component composition. The composition of components and services determines the feasible interaction ; therefore we present the various cases of interaction between services which are the basis of composition and interaction. We present the formal analysis methods and the COSTO toolbox that accompany the Kmelia approach. The article is illustrated with the CoCoME example which deals with the management of a remote sale system.*

MOTS-CLÉS : *Composants, Services, Architecture Logicielle, Correction, Assertions*

KEYWORDS: *Components, Services, Software Architectures, Correctness, Assertions*

1re soumission à *TSI : Composition des objets, composants, services*, le 1er décembre 2009.

1. Introduction

Contexte. La composition de modules ou composants pour former des systèmes de grande taille est une démarche éprouvée dans divers domaines techniques. La modularité comme principe fondamental de construction est aussi adoptée pour la construction de programmes et de logiciels. Cependant le niveau d'abstraction des éléments composés est variable d'un langage à un autre et d'une méthode à une autre. Ainsi on peut composer des fonctions, des modules, des objets, des composants, des services, etc. Le développement basé sur l'emploi de composants met l'accent sur l'approche de composition : la réutilisation et la composition de composants, élémentaires ou non, sont au coeur de cette approche ; les composants peuvent être de niveaux d'abstraction variés ; leurs contextes d'exécution peuvent être centralisés ou répartis. De nombreux défis restent à relever pour une pratique courante de cette approche.

Parmi les défis courants des approches à composants, on peut citer le besoin de langages expressifs pour la description et l'assemblage (composition) de composants, l'interopérabilité entre composants, la construction de composants corrects, fiables et réutilisables.

Dans ce contexte, nous avons entrepris l'élaboration d'une approche de construction de composants *corrects* nommée Kmelia. Notre approche considère initialement un modèle à composants abstraits et formels afin de pouvoir raisonner rigoureusement sur les composants construits. Le modèle est assorti d'un langage de spécification de composants appelé aussi Kmelia. Les particularités de l'approche Kmelia sont : c'est un modèle formel à composant qui utilise très peu de concepts (service, composant, assemblage) ; les composants sont multi-services ; les services peuvent être hiérarchisés. Pour assurer la correction des composants et de leur assemblage nous nous appuyons sur des techniques formelles.

Dans la pratique industrielle des composants, de nombreuses limitations existent : il n'existe pas d'approche largement admise qui propose la construction systématique des composants ; de nombreuses approches opèrent directement au niveau de l'implantation sans référer aux niveaux importants de la spécification et de la définition des propriétés souhaitées ; par exemple les approches à la UML agrègent un ensemble de diagrammes qui couvrent la conception, l'implantation, etc, sans pouvoir assurer ni leur cohérence globale ni l'adéquation avec le code développé par la suite.

Motivations. Notre travail s'attaque à lever certaines des limitations de l'approche par composants ; nous visons à rendre systématique et ainsi à simplifier la construction par composants en utilisant les approches formelles dès les premières phases de la construction de logiciels ; on peut ainsi construire des composants dont on peut certifier, par des preuves formelles, la correction vis à vis des spécifications et qu'on peut maintenir plus facilement en remontant à leurs spécifications.

Contributions. Dans cet article, nous présentons une synthèse des travaux et résultats autour de l'approche Kmelia pour le développement par composants. Ces travaux ont été faits progressivement et ont donné lieu à des résultats publiés successivement

(Attiogbé, André et Ardourel 2006), (André, Ardourel et Attiogbé 2007c), (André, Ardourel et Attiogbé 2007), (André, Ardourel et Attiogbé 2008). En dehors de l'aspect synthétique de cet article, nous mettons en exergue la composition dans le modèle Kmelia. Nous avons fait le choix de classer les possibilités de composition en deux catégories ; l'une verticale où on structure en profondeur et l'autre horizontale où on structure en largeur. Compte tenu des caractéristiques de composition, nous présentons les différentes formes d'interaction des services dans le modèle à composants Kmelia.

Plan de l'article. Dans une première partie, sections 2 et 3, nous présentons les principales caractéristiques du modèle à composants Kmelia, puis la composition comme concept de structuration des composants pour construire des modèles de grande taille. Dans une seconde partie, nous traitons des interactions induites par la structuration des composants et services (section 4) et les méthodes de vérification élaborées pour assurer la correction des composants et de leurs assemblages (section 5). L'article se termine par un positionnement de Kmelia par rapport à d'autres approches à composants et les perspectives d'évolutions de nos travaux.

2. Kmelia : un modèle à composants multi-services

Kmelia est un langage et un modèle à composants multi-services. Les services décrivent des fonctionnalités avec la possibilité d'interagir avec d'autres services. Les composants sont abstraits et formels de façon à pouvoir y exprimer des propriétés et à les vérifier (Attiogbé *et al.* 2006).

La proposition de Kmelia répond aux besoins suivants : un modèle à composants où les interfaces sont plus riches que les signatures des services offerts, permettant ainsi une réelle réutilisation des composants ; un modèle indépendant des plateformes d'exécution ; un langage expressif de spécification de composants et de leur assemblage ; un cadre de développement de composants où on peut élaborer des composants corrects par construction et raisonner sur des assemblages de ceux-ci.

Kmelia peut être utilisé pour décrire des logiciels sous la forme d'assemblages abstraits de composants ; ils peuvent alors être implantés dans un environnement centralisé ou dans un environnement distribué.

Le modèle Kmelia partage des caractéristiques communes avec les approches à composants (Allen et Garlan 1997, Medvidovic et Taylor 2000) : les composants, les services et les assemblages. Mais Kmelia se distingue d'autres approches par certaines caractéristiques spécifiques. Les composants Kmelia sont abstraits, indépendants de leur environnement et par conséquent non exécutables. Kmelia permet de modéliser des composants logiciels, des architectures logicielles (assemblages) avec leur propriétés. La hiérarchisation des services et des composants est une autre caractéristique de Kmelia. La hiérarchisation est basée sur l'encapsulation et elle permet une bonne

lisibilité, la flexibilité et une bonne traçabilité dans la conception des architectures (André, Ardourel et Attiogbé 2006a).

Puisqu'il est abstrait, Kmelia peut servir de modèle commun pour l'étude de propriétés (interopérabilité, composabilité) de modèles à composants et services. Les modèles peuvent être raffinés vers des plate-formes d'exécution.

Dans la conception de Kmelia, nous avons d'abord un modèle de base (Attiogbé *et al.* 2006) comme un noyau, avec très peu de concepts (composant, service, assemblage). Ce modèle de base a été successivement enrichi avec une couche **protocole** (André *et al.* 2007c) permettant de définir des enchaînements licites de services. Ensuite une extension aux **services partagés** (induisant des canaux multipoints) et à la communication multiple est proposée dans (André *et al.* 2008). Cette extension repose sur la spécialisation des services.

2.1. *Les composants dans Kmelia*

Un **composant type** est défini par un espace d'états, une liste de services et une interface I. Un **composant** est un exemplaire d'un composant type.

L'espace d'état est un ensemble de constantes et de variables typées, contraintes par un invariant. Dans l'interface d'un composant on distingue les *services offerts* qui réalisent des fonctionnalités et les *services requis* qui déclarent les besoins du composant.

Formellement un composant type C est un 8-uplet $\langle \mathcal{W}, \mathcal{A}, \mathcal{N}, \mathcal{M}, \mathcal{I}, \mathcal{D}, \nu, \mathcal{CS} \rangle$ avec :

– $\mathcal{W} = \langle T, V, type, Inv, Init \rangle$ l'espace d'états où T est un ensemble de types, V un ensemble de variables, $type : V \rightarrow T$ une fonction de typage des variables, Inv est un invariant défini sur V et $Init$ est l'initialisation des variables de V ;

– $\mathcal{A}$ est un ensemble fini d'actions élémentaires ;

– $\mathcal{N}$ un ensemble fini de noms de service. Soient $\mathcal{N}^P$ (les services offerts) et $\mathcal{N}^R$ (les services requis) deux ensembles finis disjoints de noms[1] : $\mathcal{N} = \mathcal{N}^P \uplus \mathcal{N}^R$.

– $\mathcal{M}$ est un ensemble fini d'identifiants de messages.

– $\mathcal{I} = \mathcal{I}^P \uplus \mathcal{I}^R$ est l'interface du composant : c'est l'union de deux ensembles finis disjoints de noms $\mathcal{I}^P$ et $\mathcal{I}^R$ tels que $\mathcal{I}^P \subseteq \mathcal{N}^P \wedge \mathcal{I}^R \subseteq \mathcal{N}^R$.

– $\mathcal{D}$ est l'ensemble de descriptions de services ; il inclut les services offerts ($\mathcal{D}^P$) et les services requis ($\mathcal{D}^R$).

– $\nu : \mathcal{N} \rightarrow \mathcal{D}$ est une fonction qui associe des descriptions aux noms de services. Il y une projection de la partition $\mathcal{N}$ sur son image par ν :
$s \in \mathcal{N}^P \Rightarrow \nu(s) \in \mathcal{D}^P \wedge s \in \mathcal{N}^R \Rightarrow \nu(s) \in \mathcal{D}^R$

1. $\uplus$ denote l'union disjointe d'ensembles

– $\mathcal{CS}$ est un ensemble de contraintes relatives à l'utilisation des services de l'interface de C.

Portée des variables et observabilité de l'espace d'état d'un composant

Dans le but de permettre la conception et la composition des composants de façon indépendante des contextes précis d'utilisation, nous avons introduit dans Kmelia une notion d'*observabilité* de l'état des composants. En plus de l'interface publique d'un composant, son état peut être observé par des services ou des composants composites qui l'encapsuleraient. Par conséquent, l'ensemble des variables d'état d'un composant est partitionné en deux sous-ensembles, l'un contenant les variables observables et l'autre contenant les variables non-observables. Cette partition agit aussi sur la formation de l'invariant des composants.

2.2. *Les services dans Kmelia*

La notion de service est centrale dans Kmelia ; les services sont au cœur de la construction des composants, de leur assemblage et des interactions entre composants. Nous donnons ci-dessous leur description précise.

Un **service** modélise une fonctionnalité élémentaire ou complexe. Il est constitué d'une interface, d'une description d'état et d'assertions sous la forme de pre/post-conditions et éventuellement d'un comportement dynamique. L'interface d'un service s est une description des noms et des catégories des services dont dépend s. La catégorie pouvant être le composant auquel appartient le service voulu. La syntaxe et l'utilisation sont décrites dans (André *et al.* 2007c, André, Ardourel, Attiogbé et Lanoix 2009).

Un **service interne** d'un composant est un service offert qui n'est pas dans l'interface du composant. Il est appelable par un autre service du même composant.

Nous qualifions de **sous-service** un service offert d'un composant déclaré dans l'interface d'un autre service du même composant et appelable au sein de ce service. Le déroulement d'un sous-service se situe dans le déroulement du service qui le déclare.

Service requis et espace d'états virtuel

Dans Kmelia un service requis servR d'un composant C_r est une abstraction d'un service qui est offert par un autre composant à priori inconnu du composant C_r. Par conséquent, en plus de sa signature et de ses pre/post-conditions, le service servR possède un **contexte virtuel**. Nous avons introduit dans Kmelia la notion d'*espace d'états virtuel* $v\mathcal{W}$ pour pouvoir abstraire un service de son contexte de définition qui est un composant.

L'espace d'états virtuel caractérisant un composant C dans un service requis du autre composant C_r, est décrit par des variables et un invariant qui sont supposés être compatibles avec les variables *observables* du composant C.

Description des services

Formellement un *service* s d'un composant type C[2] est défini par un triplet $(\mathcal{IS}, l\mathcal{W}, \mathcal{B})$ où

– L'interface du service $\mathcal{IS}$ est défini par un 6-uplet $\langle \sigma, \mu, v\mathcal{W}, Pre, Post, \mathcal{DI} \rangle$ avec

- $\sigma = \langle name, param, ptype, res \rangle$ la signature du service ; avec $name \in \mathcal{N}$, $param$ un ensemble de paramètres, $ptype : param \rightarrow T$ une fonction de typage des paramètres et $res \in T$ le résultat du service ;

- $v\mathcal{W} = \langle vT, vV, vtype, vInv \rangle$ un *espace d'état virtuel* avec vT un ensemble de types, vV un ensemble de variables, $vtype : vV \rightarrow vT$ une fonction de typage des variables du contexte et $vInv$ un invariant sur vV ;

- μ un ensemble de signatures de messages $\langle mname, mparam, mptype \rangle$; $mname \in \mathcal{M}$, $mparam$ et $mptype$ sont les mêmes paramètres et types que dans la signature du service ($param$ et $ptype$) ;

- Pre est une pre-condition définie sur l'union ($\cup$) des variables dans V, vV, et $param$: $V \cup vV \cup param$;

- $Post$ est une post-condition définie $V \cup vV \cup param \cup \{$ result $\}$;

- $\mathcal{DI}$ est la dépendance du service (*service dependency*) ; elle est composée des services dont dépend le service courant. $\mathcal{DI}$ est un 4-uplet $\langle sub,\ cal,\ req,\ int \rangle$ d'ensemble disjoints : $sub \subseteq \mathcal{N}^P$ (resp. $cal \subseteq \mathcal{N}^R$, $req \subseteq \mathcal{N}^R$, $int \subseteq \mathcal{N}^P$) contient les noms des services offerts (resp. ceux requis de l'appelant, ceux requis de n'importe quel composant, les services internes) dans le cadre d'un appel à s.

– $l\mathcal{W} = \langle lT, lV, ltype, lInv, lInit \rangle$ est l'espace d'état local où lT est un ensemble de types, lV un ensemble de variables locales, $ltype : lV \rightarrow lT$ une fonction de typage des variables locales, $lInv$ un invariant d'état local défini sur lV (peut être réduit à $lInv = true$) et $lInit$ est l'initialisation des variables de lV.

– Le comportement $\mathcal{B}$ d'un service s est un système de transitions étendu (*extended labelled transition system* - eLTS), detaillé dans (Attiogbé *et al.* 2006, André *et al.* 2007c, André *et al.* 2008). Une étiquette de transition est une combinaison d'actions ; elle peut être gardée. Les actions sont soit des *actions élémentaires* de $\mathcal{A}$ soit des *actions de communication* (pour appeler ou terminer un service, pour envoyer ou recevoir un message).

Un déroulement d'un service est une trace de son système de transition. Lorsqu'il y n'y a pas de communication, avec un autre service, les transitions se déroulent comme elles sont décrites et selon l'état du composant. Les transitions sont atomiques, mais

2. et par conséquent un service d'un composant c de type C, noté $c : C$.

leurs enchainements peuvent être interrompus. Le comportement du service est alors entrelacé avec d'autres services.

2.3. *Les assemblages dans Kmelia*

Les composants Kmelia peuvent être assemblés ou composés via des liens entre services.

Dans un **assemblage**, les services requis par certains composants sont liés (connectés) aux services offerts par d'autres composants. Ces liaisons, appelées **liens d'assemblage**, établissent des canaux abstraits implicites pour les communications entre services. Dans le modèle de base les canaux sont point-à-point et bidirectionnels. La figure 1 illustre une vue partielle d'un assemblage pour une application de commerce électronique dans laquelle on se focalise sur le processus de vente `process_sale` pour lequel deux sous-services sont requis (`ask_amount` et `bar_code`). Il n'y a pas de restrictions à la profondeur des sous-services et sous-liens d'un assemblage.

Un (composant) **composite** (identifié syntaxiquement par le mot-clé **composition** dans le langage Kmelia) est un assemblage qui est encapsulé dans un composant. Dans cette forme de composition détaillée en section 3.4 on peut *promouvoir* des variables et des services des composés vers le composite.

2.4. *Un exemple de spécification en Kmelia*

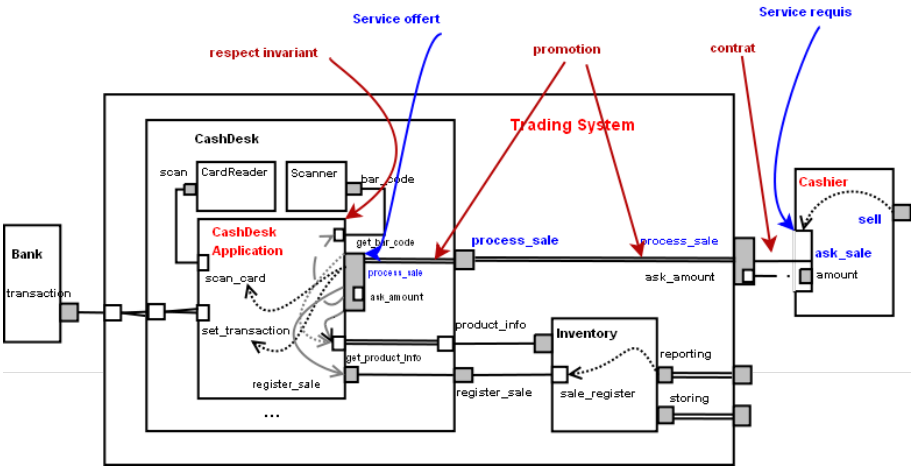

Figure 1 – Description Kmelia simplifiée du système CoCoME.

Nous illustrons à présent les notions de base du langage Kmelia par une partie du cas CoCoME (Common Component Modelling Example (Rausch, Reussner, Miran-

dola et Plasil 2008)). Il s'agit d'un système de vente à distance sous forme d'une collection de composants (caisse, scanner, imprimante, lecteur de carte...) inter-connectés et qui interagissent. Il inclut des fonctionnalités directement liées aux achats du client (la lecture optique des codes de produits, le paiement par carte ou en espèces, etc.) et d'autres tâches telles que la gestion du stock et la génération des rapports, etc.

Nous retenons de l'étude de cas un extrait de la modélisation qui illustre les principaux concepts de Kmelia : des composants (`CashDeskApplication` ou `Inventory`) reliés par leur services dans un assemblage ; le service offert `process_sale(id)` est lié au service requis `ask_sale(id)`. L'assemblage global est mis en évidence dans la figure 1.

Nous nous appuyons sur le composant *CashDeskApplication*. L'état du composant est précisé dans le listing 1.

Listing 1 – Kmelia specification CashDeskApplication

```
COMPONENT CashDeskApplication
INTERFACE
   provides : {process_sale , register_sale}
   requires : {get_product_info, set_transaction, scan_card}
USES {COCOMELIB}
TYPES
   SALE_STATE :: enum {open, close}
CONSTANTS
   null : Integer := 0
VARIABLES
   list_id : setOf Integer;
   state : SALE_STATE
  # ...
SERVICES
#--------- provided services ---------
provided process_sale(id : Integer) : Boolean
  //...
End
provided register_sale(item : OrderTO; prod : ProductTO) : Boolean
  //...
End
#--------- required services ---------
required get_product_info(prod_id : Integer) : ProductTO
End
required scan_card() : String
End
required set_transaction(credit_info : String)
End
required get_bar_code() : Integer
End
required ask_amount() : Integer
End
END_SERVICES
# end of Cash_Desk_Application
```

Le composant *CashDeskApplication* offre un service vente *process_sale*. Ce dernier fait appel à d'autres services tels que *bar_code()* vers son service appelant, ou bien *set_transaction*, *get_product_info* vers d'autres composants. Le service *process_sale* est lié au service requis *ask_sale*, il en est un fournisseur. Cette dépendance entre services est détaillée dans la spécification de l'interface du service. Illustrons ceci par

l'exemple du service *process_sale* du listing 2 extrait de la spécification Kmelia du composant *CashDeskApplication*.

Listing 2 – Kmelia specification CashDeskApplication

```
provided process_sale(id : Integer) : Boolean
Interface
   // subprovides : {}
   calrequires : {ask_amount}
   extrequires : {get_bar_code,get_product_info, set_transaction, scan_card}
   intrequires : {register_sale}
 Pre
    (id in list_id)&&(state = open)
Variables
   prod_id, total, amount_var, rest : Integer;
   authorisation : Boolean;
   credit_info : String;
   prod_info : ProductTO;
   order : OrderTO;
   payment_mode : PaymentMode
Behavior
   Init i # initial state
   Final f # final state
{    i — total := 0 —> e0,
   e0 — __CALLER ? new_code() —> e1,
   e1 — prod_id := _get_bar_code !! get_bar_code() —> e2,
   e2 — prod_info :=_get_product_info !! get_product_info(prod_id) —> e3,
   e3 — { sum(prod_info, total) ; display(prod_info)} —> e4,
   e4 — __CALLER ? new_code() —> e1,
   e4 — __CALLER ? endSale()  —> e5,
   e5 — {order.id := getOrderId();
                        order.deliveryDate := getDate();
                        order.orderingDate := getDate()
        } —> e6,
   e6 — __CALLER ? payment(payment_mode) —> e7,
   e7 — [payment_mode = cash] display ("cash payment")—> e8,
   e7 — [payment_mode = card] display("card payment") —> e13,
   e8 — amount_var := __CALLER??ask_amount(total) —> e9,
   e9 — [amount_var < total] __CALLER!! process_sale(false) —> f,
   e9 — [amount_var >= total] rest := amount_var – total  —> e10,
   e10 — __CALLER ! rest_amount(rest) —> e11,
   e11 — __SELF!! register_sale(order,prod_info) —> e12,
   e12 — __CALLER!! process_sale(true) —> f,
   e13 — credit_info := _scan_card!! scan_card() —> e14,
   e14 — _set_transaction!! set_transaction(credit_info) —> e15a,
   e15a — _set_transaction ! set_transaction(total) —> e15,
   e15 — _set_transaction ? get_authorisation(authorisation)—> e16,
   e16 — [authorisation] _set_transaction ! debitAccount()—> e11,
   e17 — [not authorisation] __CALLER!! process_sale(false) —> f
}
Post
   state = open
End
```

Une description du service de vente *sell* modélisant le comportement du composant Cashier figure dans le listing 3. Dans l'état *e4* la notation entre chevron indique que le sous-service *amount* peut être invoqué dans cet état.

Listing 3 – Kmelia specification Cashier

```
provided sell()
Interface
   subprovides : {amount}
   extrequires : {ask_sale}
```

```
Variables # local to the service
   id : Integer;
   mode : PaymentMode;
   money : Integer;
   sale_success : Boolean
Behavior
   Init i
   Final f
{   i --  {display("New sell, please enter your cashier identifier ");
       id := readInt()     # call an internal action
       } --> e10,
   e10 -- _ask_sale!!ask_sale(id) --> e11,
   e11 -- _ask_sale ! start_sale() --> e12,
   e12 -- _ask_sale ! end_Sale()  --> e13,
   e12 -- _ask_sale ! new_code() --> e12,
  i --  {display("please choose your payment mode");
       mode := readPaymentMode()     # call an internal action
       } --> e10,
      e13 -- _ask_sale ! payment(mode) --> e14,
   e14 <<amount>>,        # subservice provided in state e4 only
   e14 -- [mode = cash]_ask_sale ? rest_amount(money) --> e14,
   e14 -- _ask_sale??ask_sale(sale_success) --> f
}
End
```

Listing 4 – Kmelia specification Cashier

```
provided amount() : Integer
Variables # local to the service
   value : Integer
Behavior
   Init i    # initial state
   Final f   #  final state
{
   i --  value := readInt() --> e1,
   e1 -- __CALLER!!amount(value) --> f
}
End
#--------- required services ---------
required ask_sale(cashier_id : Integer) : Boolean
End
```

Dans la suite nous nous attachons à décrire les caractéristiques de la composition dans Kmelia.

3. Composition dans le modèle Kmelia

La composition est l'opération générique fondamentale de construction des composants et systèmes. Elle prend diverses formes en fonction du type de construction visée. On peut imbriquer un service/composant dans un autre : composition verticale ; on peut lier un service/composant à un autre pour une opération de communication par exemple : composition horizontale. Dans la suite nous détaillons les différents pris en compte dans Kmelia.

3.1. *Composition verticale de services*

Kmelia permet l'expression de services complexes et leur composition. La composition verticale de services repose sur l'ajout de *points d'expansion* (aux états ou aux actions sur $\mathcal{B}_s$) qui permettent d'inclure un service dans un autre et qui sont expansés lors des vérifications de compatibilité de services. Un appel de sous-service au sein d'un service s est évalué dans le contexte de s. Différents types de points d'expansion sont utilisés selon que l'on désire un appel *obligatoire* ou *optionnel* de service.

Soit la description suivante d'un service offert (servP) :

```
provided servP()
Interface
   subprovides : {subServ1, subServ2}
...
Behavior
   Init i
   Final f
{   i --  a1 --> e1,
   e1 -- [[subServ2]] --> e2,
   e2 <<subServ1>>,     # sous-service offert dans l'état 2
   e2 -- a2 --> e3,
   e3 -- a3 --> f,
}
End
```

Dans cette description subServ1 et subServ2 sont des sous-services de servP, c'est-à-dire des services offerts dans le cadre de servP. Le sous-service subServ1 peut être invoqué dans l'état e2 uniquement, il est *optionnel*, il peut alors appelé par l'appelant de servP. A la fin du déroulement de subServ1, le contrôle revient dans l'état e2 ; il est possible d'itérer l'appel de subServ1. Le sous-service subServ2 lui, doit être invoqué dans l'état e1, il est *obligatoire*, (il doit être appelé par l'appelant de servP). Tout se passe dans ces deux cas comme si on descendait en profondeur dans les systèmes de transition par rapport à un noeud du système de transition du service initial ; d'où la composition verticale.

Il y a une variante de chacun de ces deux cas de composition verticale. Une variante du cas optionnel (notée $<|$subServ1$|>$) et une du cas obligatoire (notée [| subServ1 |]) qui proposent une inclusion des sous-services sans appel de service de la part de l'appelant, ni de retour de valeurs. Les systèmes de transition sont alors composés par dépliage de l'un dans l'autre au niveau des noeuds ou transitions.

Un exemple est donné avec le cas CoCoME pour la description du service de vente *sell* du listing 3. Le sous-service amount (listing 4 est invocable de manière facultative dans l'état *e4*.

Nous nous servons de la composition verticale pour définir des *protocoles* dans Kmelia c'est-à-dire des enchaînements licites de services. Les protocoles sont, dans notre proposition, des modes d'emploi pour les composants. Nous avions détaillé ces résultats dans (André *et al.* 2007c).

3.2. *Dépendance et liaison de services*

La composition de composants dans Kmelia fait appel à des liens entre services. Commençons par définir formellement[3] la notion de liens.

Soit $\mathcal{C}$ un ensemble de composants $c_k : C_k$ pour $k \in 1..n$ avec $C_k = \langle\langle T_k, V_k, type_k, Inv_k, Init_k\rangle, \mathcal{A}_k, \mathcal{N}_k, \mathcal{M}_k, I_k, \mathcal{D}_k, \nu_k, \mathcal{C}_{Sk}\rangle$ comme défini dans la section 2.1. Soit $\mathcal{N}$ l'ensemble des noms de services de $\mathcal{C}$ ($\mathcal{N} = \bigcup_{k\in 1..n} \mathcal{N}_k$).

La dépendance de services $depends_k$ d'un composant $c_k : C_k$ est la généralisation des dépendances de services externes de $\mathcal{D}I$ au niveau des composants :

$$
\begin{array}{l}
depends_k : \mathcal{N}_k \leftrightarrow \mathcal{N}_k \\
\forall (n, m) : depends_k \bullet (\exists sm : \mathcal{N}_k \bullet \\
\qquad (n \in cal_{sm}) \vee (n \in req_{sm}) \vee (n \in sub_{sm}))
\end{array}
$$

Un lien entre deux services est défini par un quadruplet de noms de composants et de services avec les restrictions suivantes : (1) les noms de service sont ceux de leur composant-type, (2) un service n'est pas lié à lui-même.

$$
\begin{array}{ll}
BaseLink : I\!P\ (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N}) & \\
(1) & \forall (c_i, n_1, c_j, n_2) : BaseLink \bullet n_1 \in \mathcal{N}_i \wedge n_2 \in \mathcal{N}_j \\
(2) & \forall c_i : \mathcal{C},\ n_1 : \mathcal{N}_i \bullet (c_i, n_1, c_i, n_1) \notin BaseLink
\end{array}
$$

Un sous-lien est un lien défini dans le contexte d'un autre lien. La notion de sous-lien est relative à la fois à la dépendance de service sur l'appelant (ensemble cal de $\mathcal{D}I$) et celle des sous-services (ensemble sub de $\mathcal{D}I$). Il n'y a pas de dépendance circulaire entre liens.

$$
\begin{array}{ll}
SubLink : BaseLink \leftrightarrow BaseLink & \\
(1) & \forall (l_1, l_2) \in SubLink \bullet (l_2, l_1) \notin SubLink^*
\end{array}
$$

où $A \leftrightarrow B$ désigne une relation entre les ensembles A et B et $SubLink^*$ est la fermeture transitive de la relation $SubLink$.

Dans Kmelia les liens et sous-liens apparaissent sous forme de **liens d'assemblage** dans la composition horizontale et de **liens de promotion** dans la composition verticale de composants.

3.3. *Composition horizontale*

Partant d'un ensemble de composants, la composition horizontale est l'opération qui consiste à construire un assemblage en reliant les services de ces composants. L'*assemblage de composants* en Kmelia correspond à l'architecture de composants

3. On utilise ici une notation « à la Z ou B » pour décrire les ensembles et relations.

dans d'autres modèles (Shaw et Garlan 1996, Bures, Hnetynka et Plasil 2006). Les connecteurs sont simplement des liens d'assemblage (liaisons dans (Bruneton, Coupaye, Leclercq, Quéma et Stefani 2006, Bures *et al.* 2006)). La spécificité dans Kmelia est qu'on relie des services et non des composants ou des interfaces. De plus un lien peut définir une structure hiérarchique de sous-liens qui doit être cohérente avec la composition verticale des services : les sous-services offerts dans le cadre d'un service apparaissent dans les sous-liens.

3.3.1. *Assemblage de composants*

Un *assemblage type* est un ensemble de composants liés par leurs services. Un *assemblage* est un exemplaire d'un assemblage-type[4]. Formellement, un assemblage-type de composants est un triplet $A = (\mathcal{C},\ alinks,\ subs)$ où $\mathcal{C}$ est un ensemble de composants $c_k : C_k$ pour $k \in 1..n$, $alinks$ est un ensemble de liens d'assemblage entre services de $\mathcal{C}$ et $subs$ est une relation d'inclusion de liens.

Liens d'assemblage

Un *lien d'assemblage* entre un service n_1 d'un composant C_1 et un service n_2 d'un composant C_2 est une abstraction d'un canal de communication reliant n_1 à n_2. Chacun des services n_1 et n_2 est dans une des interfaces des composants.

$$
\begin{array}{ll}
& alinks \subseteq BaseLink \wedge \\
(1) & \quad (\forall (c_i, n_1, c_j, n_2) : alinks \bullet c_i \in \mathcal{C} \wedge c_j \in \mathcal{C} \wedge \\
(2) & \qquad ((n_1 \in \mathcal{I}_i^P \wedge n_2 \in \mathcal{I}_j^R) \vee (n_1 \in \mathcal{I}_i^R \wedge n_2 \in \mathcal{I}_j^P))) \\
& subs \subseteq SubLink \wedge \\
(3) & \quad (\mathrm{dom}\, subs - \mathrm{ran}\, subs) \subseteq alinks \wedge \\
(4) & \quad (\forall ((c_i, n_1, c_j, n_2) \mapsto (c_k, n_3, c_l, n_4)) \in subs \bullet c_i = c_k \wedge c_j = c_l) \wedge \\
(5) & \quad (\forall (c_i, n_1, c_j, n_2) : \mathrm{ran}\, subs \bullet ((\nu_i(n_1) \in {\mathcal{D}^P}_i) \ \mathrm{xor}\ (\nu_j(n_2) \in {\mathcal{D}^P}_j)))
\end{array}
$$

Les composants des liens sont les composants de l'assemblage (1). Il y a symétrie *requis-offert* dans un lien (2). Les sous-liens dépendent *in-fine* des liens d'assemblage de plus haut niveau (3) et concernent les mêmes composants (4). Les services offerts sont liés aux services requis (2 et 5).

Correspondances de contexte dans les liens d'assemblage

Dans le cadre des liens d'assemblage, on doit établir deux correspondances afin d'assurer la cohérence des liens et donc de l'assemblage global. La première correspondance établit explicitement la relation entre le contexte virtuel (cf section 2.2) du service requis (si un tel contexte est défini) et le contexte observable du composant du service offert : c'est la correspondance de contexte (*context mapping* dans(André *et al.* 2009)).
Considérons un service requis sr d'un composant cr de type CR, lié à un service offert sp d'un composant cp de type CP. Les variables de l'espace d'état virtuel (vV_{sr})

4. Comme d'habitude, si A est un assemblage type, a :A désigne un assemblage a de type A

de sr sont mises en correspondance avec les variables *observables* de cp (V^O_{CP}) par une fonction (totale) $vmap : vV_{sr} \rightarrow exp(V^O_{CP})$ où $exp(X)$ désigne une expression sur les variables de X.

La seconde correspondance permet de relier explicitement les identifiants des messages utilisés dans les services liés : c'est la correspondance de messages (*message mapping* dans (André *et al.* 2009)). Les paramètres des messages doivent conserver leur ordre ; dans le cas contraire il s'agit d'un problème d'adaptation que nous avons traité dans (André, Ardourel et Attiogbé 2007b). Formellement, chaque nom de message de sr est associé à un nom de message de sp par la fonction totale $mmap : mname_{sr} \rightarrow mname_{sp}$. Si les services établissent des dépendances de type subprovides et calrequires alors des sous-liens sont à définir dans le cadre de ce service. Leur spécification est similaire à celle des liens. Soit l'assemblage suivant :

Listing 5 – Assemblage de composants et services

```
Assembly
  Components
    cp : CP;
    cr : CR
  Links // ————————assembly links————————
    @la: p–r cp.provServ cr.reqServ
       context mapping
         cr.var1 = expr(cp.varA, cp.varB...),
         cr.var2 = expr(cp.varA, cp.varB...),
       message mapping
         provServ.msg1 = reqServ.msgA
         provServ.msg2 = reqServ.msgB
       sublinks : {lasub}
    // ————————sublinks————————————
    @lasub: r–p cp.subReq cr.prov
    ...
End // assembly
```

Dans cette spécification d'assemblage, deux composants cp et cr sont assemblés par un lien d'assemblage @la indiquant que le service requis reqServ de cr est *réalisé* par le service offert provServ de cp. Le préfixe **p**−r indique le sens de lecture du lien. Par la correspondance de contexte, la variable var1 du contexte virtuel de reqServ s'exprime en fonction des variables varA, varB... de l'espace d'état observable du composant cp. Par la correspondance de messages, les messages nommés msg1 dans provServ sont nommés msgA dans reqServ. Les sous-liens sont l'expression de la correspondance des dépendances de services. Ainsi si le service offert provServ requiert le service subReq dans sa dépendance `calrequires` alors un sous-lien doit être défini vers un service prov offert directement dans l'interface offerte du composant cr ou dans la dépendance `subprovides` du service reqServ.

Notons que les services apparaissant dans la dépendance `extrequires` du service offert provServ nécessitent la spécification d'un service requis du composant cp et par là-même un lien vers un autre serveur (du service offert). Notons que les services apparaissant dans la dépendance `intrequires` du service offert provServ nécessitent la spécification d'un service offert du composant cp et aucun lien n'est explicité.

3.3.2. *Illustration*

Reprenons l'exemple CoCoME. La figure 1 est une modélisation partielle qui se focalise sur le système *Trading System* de gestion des ventes, composé de *Inventory* un composant de gestion de persistance et de *CashDesk* un composant de gestion de caisse. Ce dernier composant offre un service de vente *process_sale* en se basant sur un composant applicatif *CashDeskApplication* et des composants modélisant les périphériques d'entrée/sortie.

Dans la figure 1, les traits simples entre services sont des *liens d'assemblage* qui associent des services offerts à des services requis (un service requis est « réalisé » par un service offert).

Listing 6 – Kmelia specification Le modèle Kmelia du système CoCoMe

```
COMPONENT CoCoMESystem
  //main assembly
INTERFACE
  provides : {sell}
SERVICES
END_SERVICES
COMPOSITION
  Assembly
    Components
      ts : TradingSystem;
      b : Bank;
      c : Cashier
    Links //////////assembly links/////////
      @trans: r–p ts.set_transaction b.transaction
      @sale: p–r ts.process_sale c.ask_sale
          sublinks : {lamount}
      //////////promotion sublinks/////////
      @lamount: r–p ts.ask_amount c.amount
  End // assembly
  Promotion
    Links //////////promotion links/////////
      @sell: p–p c.sell SELF.sell
END_COMPOSITION
```

3.3.3. *Assemblage multiple, Service partagé et liens n-aires*

Dans ce qui précède, nous avons considéré des assemblages simples : (1) chaque exemplaire d'un composant-type est nommé par une variable (2) les liens sont exclusivement entre deux composants (de type `1-1`. L'exclusivité du lien signifie que si n services sont liés à un service, alors on considère réellement n liens indépendants.

Cependant le modèle a été étendu dans (André *et al.* 2008) pour autoriser plusieurs exemplaires d'un composant type (tableau de composants) et des services partagés pour permettre les interactions multi-parties (liens `1-n` ou `n-1`).

Un tableau de composants de même type C est déclaré par c[n]:C, où n est le nombre maximum d'exemplaires. Chaque composant est accessible par son indice dans le tableau c[i]

Le modèle Kmelia offre la possibilité de partager des services. Un *service offert partagé* est un service lié à plusieurs services requis de plusieurs composants de types

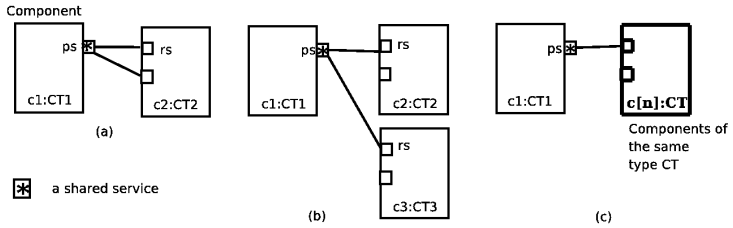

Figure 2 – Assemblages avec des services offerts partagés

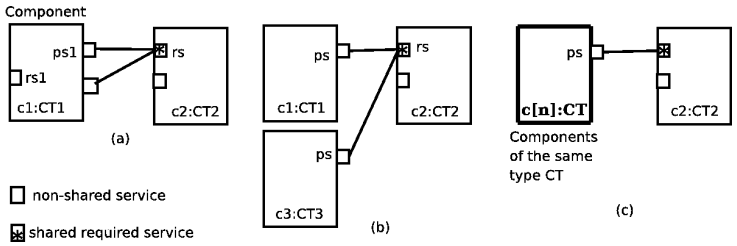

Figure 3 – Assemblage avec des services requis partagés

quelconques (lien de type `1-n`). Il peut donc être invoqué par plusieurs appelants. Par défaut les appelants sont rangés dans un tableau, ils ont le même rôle dans la collaboration mais il est possible de leur faire jouer des rôles différents. Les appelants n'ont *a priori* aucune connaissance les uns des autres. Par exemple, un serveur de messagerie instantanée offre un service partagé de conférence à plusieurs clients.

Un *service requis partagé* est un service lié à plusieurs services offerts de plusieurs composants de types quelconques (type `n-1`). Il va donc mettre en concurrence plusieurs fournisseurs. Par exemple, un service Web de comparatif peut émettre des requêtes et collecter les résultats de différentes façons (le premier reçu, le meilleur selon un critère...).

3.4. *Composition verticale de composants*

La composition verticale de composants consiste en l'encapsulation d'un assemblage dans un composant, appelé **composite** par la suite. Ainsi une succession de *composition verticales* aboutit à une hiérarchie de composants. Afin de faciliter la réutilisation et le passage à l'échelle, l'encapsulation d'un assemblage dans un composite masque les composants par défaut. Le composite doit pouvoir être utilisé comme tout autre composant. Le composite peut voir et donc utiliser dans ses assertions les variables observables de ses composants directs. Il peut également utiliser les services de leurs interfaces dans les services qu'il définit. Le composite peut promouvoir certains des services et des variables observables des composants en les présentant

comme siennes. L'encapsulation pouvant se faire de manière répétée, la visibilité des espaces d'état se fait de proche en proche, chaque composant déclarant ses variables observables parmi lesquelles peuvent figurer des variables promues. De la même façon, les variables et les services y compris leurs dépendances, peuvent être promus à l'interface du composant résultant de l'encapsulation.

En guise d'illustration, prenons l'architecture du système CoCoME de la figure 1. Les emboîtements de composants dénotent la relation de composition par encapsulation de composants ; les traits doubles sur les services sont des *liens de promotion* ; un service d'un composant est promu au niveau du composite qui le contient.

Dans la spécification textuelle, les liens de promotion sont une section de la Promotion. Par exemple, dans le listing 6, le service offert sell du composant c : Cashier est promu au niveau du système CoCoMESystem, dont il devient un point d'entrée. La promotion d'une variable v_1 : T_1 du composant c_1 : C_1 se fait dans la déclaration de l'espace d'état du composite C_2 par v_2 : T_1 FROM c_1.v_1. Le renommage est facultatif.

3.5. *Bilan*

En résumé, nous avons présenté dans cette section, les principes de la composition dans Kmelia. La composition verticale au niveau des services contenus dans un composant : un service peut comporter des points d'expansion, qui sont des noeuds ou des transitions annotées selon un formalisme approprié, où s'expansent un autre service indiqué dans l'annotation. La composition verticale au niveau des composants où un composant peut être encapsulé dans un autre avec plusieurs niveaux. La composition horizontale de composants résulte en l'assemblage de composants ; elle se fait via la liaison de services de composants différents ; les services liés peuvent ainsi interagir et par conséquent assurer l'interaction entre les composants. La liaison des services est vue comme la composition horizontale de services : c'est une composition parallèle ou les services évoluent de façon indépendante ou communiquent via le canal abstrait établi par leur liaison.

4. Interaction multi-parties dans Kmelia

Plusieurs formes d'interaction sont possibles entre les parties d'un système à base de composants. Dans le cas de l'approche Kmelia, les interactions autorisées sont étroitement liées à la composition des composants. Par conséquent les interactions sont vues sous différents angles. Il peut y avoir des interactions entre les services d'un même composant selon la composition verticale (interaction intra-composant) ; les interactions entre les services de composants encapsulés (interaction intra-composant) ; la composition entre les services dans une composition horizontale (interaction inter-composant).

Une simple interaction peut avoir lieu entre deux services provenant chacun d'un composant différent. Les systèmes de transition qui modélisent les comportements des services sont le support de l'interaction.

Les interactions démarrent à partir du déroulement d'un service. Un service offert d'un composant, lorsqu'il est appelé, devient actif et se déroule selon son système de transition, depuis son état initial jusqu'à son état final.

Nous avons distingué l'interaction simultanée entre plusieurs services relevant d'un ou plusieurs composants ; on parle alors d'interaction multi-parties.

4.1. *Interaction intra-composant*

Nous précisons ici, d'une part la manière dont les services, les services internes et les sous-services d'un même composant interagissent et d'autre part la manière dont les services de composants imbriqués interagissent.

Par défaut, des services d'un même composant peuvent être actifs simultanément, leur déroulement est entrelacé. Cependant, un service peut être décrit comme non-interruptible, dans ce cas son déroulement doit atteindre l'état final avant de s'arrêter.

Considérons l'interaction entre un service et un de ses sous-services. Un service d'un composant C et ses sous-services ne partagent pas leurs variables locales, mais celles du composant C.

Un service peut, lors de son déroulement, évoquer un sous-service. Le sous-service prolonge alors le déroulement de son appelant, il n'y a pas d'interaction entre eux. Un service peut communiquer avec ses sous-services à travers les variables du composant. L'appel/retour de service crée/termine cette interaction entre un service et un sous-service dans un composant.

En ce qui concerne les services internes, il peut y avoir une interaction entre un service interne et son appelant ; les interactions sont ici basées sur les canaux de communication abstraits associés aux services. Les opérateurs d'appel/retour (!! et ??) de service crée/termine cette interaction entre un service et un service interne dans un composant.

Lorsqu'un composant C_e est encapsulé dans un autre C_i, du fait des promotions dues à la composition verticale des composants, les nouveaux services ajoutés à C_e peuvent interagir avec des services offerts par des C_i. Ces interactions se font alors par l'appel des services de C_i.

4.2. *Interaction inter-composant*

L'interaction inter-composant est rendue possible par la composition horizontale. Considérons dans un premier temps des interactions basées sur des liaisons d'un service/composant à un autre service/composant.

L'interaction entre deux services venant de deux composants distincts est la base de l'interaction entre composants ; elle se fait donc de proche à proche. Un service d'un composant, pendant son déroulement, appelle un autre service qui est un requis du composant. Le déroulement se passe comme si le service requis était remplacé par le service effectif offert par un autre composant. Les deux services se déroulent en parallèle et échangent en communicant sur un canal abstrait. Les services interagissent via les communications synchrones ou asynchrones. C'est la composition horizontale qui définit le support de cette communication.

Le langage Kmelia offre des primitives de communication qui instaurent explicitement ces communications. Ces primitives de Kmelia et leurs syntaxes sont inspirées de CSP de Hoare. La forme élémentaire des actions de communication est

```
channel( !|?| !!|??) message(param*)
```

Les communications sont alors des paires d'actions complémentaires : *envoi de message(!)-réception de message(?), appel de service(!!)-attente du démarrage de service(? ?), envoi du résultat de service(!!)-attente du résultat de service(? ?).*

La forme élémentaire des actions de communication a été étendue pour exprimer diverses autres interactions comme on le verra plus loin.

Un service pouvant communiquer avec plusieurs autres services de composants différents, l'interaction entre composants se généralise à une interaction entre plusieurs composants via le réseau formé par les services communicants. En effet, le déroulement d'un service peut entrainer en cascade le déroulement et donc la communication avec d'autres services, d'où la formation d'un réseau de services communicants.

Par construction, nous éliminons la formation de boucles dans les assemblages (par composition) donc les communications ne sont pas interblocantes par la présence de cycles.

L'interaction globale dans un assemblage est au final une juxtaposition d'interactions de proche en proche. Dans la section qui suit nous présentons le cas de l'interaction entre un service et plusieurs autres.

4.3. *Interaction multi-parties*

La composition horizontale dans Kmelia autorise des liens d'un service d'un composant vers plusieurs services d'un ou plusieurs composants. Dans ce cas il y a potentiellement une interaction entre plusieurs services.

On est dans le cadre d'une généralisation des interactions entre services. Les communications et les primitives associées sont adaptées en conséquences. Dans ce contexte, les actions de communications prennent la forme :

```
channel[<selector>]( !|?| !!|??)message(param*)
```

Les valeurs de `<selector>` sont : `ALL` pour tous les services concernés par la communication, `i` pour un service donné et `:i` pour un service quelconque.

Lorsqu'un service offert servP est partagé par plusieurs services requis servR_i, alors il y a interaction simultanée entre le comportement du service servP et les comportements respectifs des services ServR_i. Cette interaction se déroule comme suit :

– les actions élémentaires de tous les services se déroulent de façon indépendante, et sont par conséquent entrelacées.

– les actions de communications utilisées dans les comportements des services forcent soit à une synchronisation entre tous les services (c'est le cas quand les opérateurs sont précédés de sélecteurs spécifiques . . . [ALL]?? . . ., . . . [ALL]!! . . .) soit à une synchronisation entre certains services uniquement (c'est le cas avec . . . [: i] ? . . .).

Le lecteur trouvera la présentation complète de ces cas de communication dans (André *et al.* 2008).

De la même manière que dans le cas précédent, lorsqu'un service requis partagé est lié à plusieurs services offerts, il y a interaction multiple simultanément ; les actions élémentaires sont entrelacées alors que celles de communication sont régies par la sémantique des opérateurs de communication.

5. Outillage et analyse formelle des spécifications Kmelia

L'élaboration de la proposition Kmelia est accompagnée par le développement d'une plateforme d'expérimentation ouverte : COSTO[5].

L'approche formelle adoptée pour Kmelia donne la possibilité d'analyser rigoureusement les modèles à composants construits. L'analyse des composants se fait à différents niveaux : le niveau des services, le niveau des composants et le niveau des assemblages.

Les composants sont analysés par rapport à des propriétés ; on peut ainsi vérifier des propriétés inhérentes aux modèles, telle que la cohérence ; des propriétés relatives aux composants telle que la cohérence d'un service, la correction d'un service par rapport à ses assertions pre-post ; des propriétés relatives à la structuration des composants telle que la composabilité ; etc.

Nous donnons ici un aperçu rapide de la plateforme logicielle associée à Kmelia puis nous présentons quelques aspects des analyses effectuées sur les spécifications Kmelia.

5. COmponent Study TOolkit

5.1. *La plateforme COSTO*

L'approche Kmelia est outillée. Les spécifications Kmelia sont analysables syntaxiquement et aussi du point de vue des propriétés. Ces analyses sont mises en œuvre dans la plateforme COSTO (André, Ardourel et Attiogbé 2007a).

La plateforme est principalement constituée

– d'un analyseur (*Type-checker*) pour les spécifications Kmelia ; l'analyseur est développé avec la technologie Java/Antlr. Nous avons également développé divers *plugins* sous Eclipse pour l'analyse à la volée, le contrôle et la coloration syntaxique. A l'issue de l'analyse d'une spécification Kmelia, un modèle interne à objet est généré en utilisant une librairie que nous avons entièrement développée pour satisfaire les besoins spécifiques à Kmelia.

– de passerelles vers MEC et LOTOS/CADP ; le but est la vérification des interactions entre les systèmes de transition qui modélisent les services. Ici nous nous sommes appuyés sur des environnements existants afin d'exploiter au mieux les résultats connus en terme d'analyse de la dynamique des systèmes.

– d'une nouvelle passerelle en cours de développement vers la méthode B.

– de divers utilitaires, pour présenter les spécifications sous forme graphique, pour générer des documents Latex des spécifications.

Dans la suite nous faisons un survol des analyses effectuées sur des composants ou assemblages Kmelia et des méthodes d'analyse que nous avons mises au point.

5.2. *Analyse de propriétés au niveau composant*

L'objectif est de vérifier la correction individuelle des composants et services avant assemblage. Pour illustrer on considère ici uniquement la propriété de cohérence des composants.

5.2.1. *Cohérence des services*

Nous vérifions la validité des assertions (pre et post-conditions du service).

Dans un premier temps nous avons choisi une approche classique qui consiste à vérifier que l'invariant d'état du composant est préservé lors du déroulement des services en respectant les préconditions : c'est la cohérence de l'invariant. Les services fournis et les services requis ne contredisent pas l'invariant de leur composant. La vérification de cette propriété de cohérence est faite à trois niveaux d'observabilité. La vérification effective se fait dans un environnement adapté ; des expérimentations sont en cours avec le langage B (Abrial 1996) ; l'idée est de générer des obligations de preuve en B qui sont ensuite prouvées. En guise d'illustration de notre méthode, voici quelques règles définies pour assurer la cohérence des services.

1) La partie observable du composant est cohérente : la partie observable de la post-condition d'un service offert est suffisante pour établir la partie observable de l'invariant :

$$\mathsf{inv(old(o))} \wedge \mathsf{pre(p,old(o))} \wedge \mathsf{post(p,old(o),o,r)} \Rightarrow \mathsf{inv(o)} \qquad \text{(INV/O)}$$

2) L'invariant en entier est préservé par les services offerts :

$$\mathsf{inv(old(o))} \wedge \mathsf{inv(old(o),old(x))} \wedge \mathsf{pre(p,old(o))} \wedge \mathsf{post(p,old(o),o,r)} \wedge \mathsf{lpost(p,old(o),o,old(x),x,r)} \Rightarrow \mathsf{inv(o)} \wedge \mathsf{inv(o,x)} \qquad \text{(INV/F)}$$

et aussi par les services internes.

$$\mathsf{inv(old(o))} \wedge \mathsf{inv(old(o),old(x))} \wedge \mathsf{ipre(p,old(o),old(x))} \wedge \mathsf{ipost(p,old(o),o,old(x),x,r)} \Rightarrow \mathsf{inv(o)} \wedge \mathsf{inv(o,x)} \qquad \text{(INV/F')}$$

3) cohérence du contexte imaginaire : chaque service requis préserve son invariant virtuel ;

$$\mathsf{invR(old(v))} \wedge \mathsf{preR(p,old(v))} \wedge \mathsf{postR(p,old(v),v,r)} \Rightarrow \mathsf{invR(v)} \qquad \text{(INV/V)}$$

Le lecteur pourra trouver une présentation complète de la méthode de vérification dans (André *et al.* 2009).

5.3. *Analyse de propriétés au niveau assemblage*

En supposant que les composants individuels aient les propriétés souhaitées, on souhaite déterminer statiquement les propriétés des assemblages. On traite ici de la cohérence et de la composabilité.

5.3.1. *Cohérence d'un assemblage*

La cohérence d'un assemblage est primordiale pour son bon fonctionnement. En dehors de la cohérence au niveau des interfaces des services qui sont reliées pour former l'assemblage, une bonne conception de l'assemblage doit permettre de détecter des incohérences au delà des simples correspondances de paramètres et types. Dans la proposition Kmelia, les assertions (pre/post-conditions) sont utilisées pour mettre en place des contrats entre les services dans les assemblages.

Nous analysons la bonne formation des assemblages en nous basons sur la notion de composabilité. De façon synthétique, quatre niveaux de compatibilité sont pris en compte :

1) Signature : le profil des services doit être compatible.

2) Dépendances : les interfaces de composants et les dépendances de services doivent être compatibles.

3) Contrat (pre/post) : le service offert "remplit le contrat" défini par le service requis.

4) Interactions : les communications entre services doivent être cohérentes et ne pas aboutir à des blocages.

Noter que ces exigences varient en fonction des éléments fournis. On autorise ainsi une certaine compatibilité et par là-même une *interopérabilité* de modèle avec des composants issus d'autres modèles (encapsulés dans des composants Kmelia). Par exemple, les composants Corba ne définissent que les signatures (dans le modèle de base).

5.3.2. *Composabilité*

Nous avons formellement défini la notion de composabilité (Attiogbé *et al.* 2006). De manière informelle, des services des composants sont composables si on peut les lier pour faire un assemblage et de telle sorte que les services interagissent. Le modèle Kmelia nous a servi de base de raisonnement pour l'analyse de la *composabilité* sur des compositions. La composabilité se ramène à la compatibilité statique et l'interopérabilité dynamique (compatibilité comportementale).

Compatibilité statique

Du fait de la description des services, la compatibilité statique lors des assemblages de composants est vérifiée de façon progressive. Une première étape de la vérification de la compatibilité statique est faite à la compilation de la spécification Kmelia par analyse des interfaces en profondeur : 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). Lorsque cette première étape de compatibilité est terminée, les assertions des services liés sont utilisées pour s'assurer du respect du contrat d'assemblage.

Préservation des contrats d'assemblage

Soit un service ServR (d'un composant C) avec les assertions PreR et PostR ; soit un service offert ServP avec les assertions PreP et PostP. Le service servP permet de satisfaire le requis ServR lorsque :

$$PreR \Rightarrow PreP \wedge PostP \Rightarrow PostR$$

De cette façon, on décrit un contrat de telle sorte que

– les composants qui peuvent être assemblés avec C sont ceux qui sont à même de respecter le contrat

– les services du composant C qui requièrent ServR savent les conditions dans lesquelles ils obtiennent des bons résultats.

En prenant en compte l'espace d'état virtuel du service requis et l'espace des variables observables du service ServP, le précédent contrat d'assemblage s'étend comme suit :

$$(Pre_R^O)_{cm} \Rightarrow Pre_P^O Post_P^O \Rightarrow (Post_R^O)_{cm}$$

avec $(Pre_R^O)_{cm}$ la précondition de ServR restreinte aux variables observables du composant qui fournit ServP ; l'indiçage avec $_{cm}$ (pour *context mapping*) indique qu'il peut être nécessaire d'appliquer une correspondance entre les variables du contexte virtuel et celles du contexte réel.

Pre_P^O (resp. $Post_P^O$) désigne la précondition (resp. la post-condition) de ServP restreinte aux variables observables.

$(Post_R^O)_{cm}$ est la postcondition de ServP restreinte aux variables observables.

Cette exigence d'analyse de cohérence au niveau des services est aussi traitée au niveau de la promotion des services requis et offerts.

Compatibilité comportementale

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 pair à pair 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 (André, Ardourel et Attiogbé 2006b) et LOTOS/CADP (Attiogbé *et al.* 2006). L'analyse de la spécification est alors faite avec ces outils. C'est la technique de *model checking* qui est la plus utilisée dans ce cas.

6. Travaux connexes

Il existe de nombreux travaux et proposition sur les modèles à composants ; ils se positionnent à différents niveaux d'abstraction et autorisent ou non le raisonnement formel pour analyser les constructions de (modèle de) systèmes. Notre approche Kmelia est plus proche des approches ou l'analyse formelle est une préoccupation de premier plan que des approches qui privilégient le codage direct. Dans cette dernière catégorie, nous mettons les propositions "industrielles" telles que JavaBeans, .net qui sont largement utilisées dans des applications réelles de grande taille. Cependant elles ne permettent pas de raisonner formellement au niveau composant ni au niveau des compositions de composants.

La démarche dans Kmelia est plus rattachée à la catégorie des modèles à composants tels que SOFA (Bures *et al.* 2006), rCOS (Chen, Liu, Stolz, Yang et Ravn 2007, Liu, Morisset et Stolz 2009), Fractal (Bruneton *et al.* 2006), BIP (Gossler et Sifakis 2005, Sifakis 2005). La composition est traitée de diverses façons selon les caractéristiques de l'approche considérée. Nous examinons quelques cas dans la suite en positionnant l'approche Kmelia par rapport à celles avec lesquelles elle est comparable.

Composition dans SOFA. Les composants SOFA élémentaires ont un comportement (*behaviour protocol*) ; les composants peuvent être composés verticalement, par encapsulation ; l'approche de structuration est descendante, c'est la méthode de conception primordiale préconisée dans SOFA ; ainsi à partir d'un composant de plus haut niveau, on descend vers les composants qui doivent y être encapsulés jusqu'aux composants de plus bas niveau. Le comportement des composants composites est semi-automatiquement compilé. Cette encapsulation des composants est comparable à la composition verticale dans Kmelia. La composition horizontale dans SOFA consiste en la composition parallèle des comportements (*protocols*) des composants concernés ; ils communiquent alors par échanges synchrones de messages. Aux spécificités des services près, c'est la même approche que pour la composition horizontale dans Kmelia. Des connecteurs, semi-automatiquement générés permettent de réaliser la composition horizontale des composants SOFA, un tel mécanisme n'existe pas encore dans Kmelia où les liaisons dues à la composition horizontale restent abstraites.

Composition dans Fractal. Même si les composants Fractal sont vus comme des entités à l'exécution, il est possible de les composer. Nous pouvons les examiner sous l'angle dimensionnel vertical ou horizontal. Les modèles Fractal utilisent des codes de programmes (Java par exemple) pour le comportement des composants ; cela limite les possibilités de composition horizontale. La composition par une liaison (*binding*) entre interface client et interface serveur est une composition horizontale ; dans ce cas, les composants Fractal peuvent communiquer par invocation d'opérations (de client à serveur), c'est la *primitive binding*. La composition par *composite binding* lie un nombre quelconque de composants ; elle se ramène en fait à plusieurs liaisons primitives. Dans les deux cas, l'analyse de la compatibilité comportementale n'est pas possible comme c'est le cas avec Kmelia. La composition hiérarchique par encapsulation des composants Fractal est comparable à la composition verticale par encapsulation de composants Kmelia ; en revanche nous n'avons pas identifié une encapsulation de niveau service comme le permet Kmelia.

Composition dans rCOS. La composition horizontale de composants Kmelia se retrouve dans rCOS sous la forme de l'union disjointe (ou composition parallèle) de composants. Cependant dans rCOS, une couche de processus donnant la dynamique des composants est ajoutée aux composants pour ordonnancer leur interaction ; en effet les services des composants rCOS sont des actions élémentaires et n'autorisent pas d'interactions contrairement aux services de Kmelia. La composition verticale est assimilable à l'opération de masquage (*hiding*) de rCOS où une union disjointe par exemple se retrouve encapsulée dans un composant. Ici la promotion des services non

masqués est systématique, à la différence de Kmelia ou l'encapsulation ne force pas les promotions des services non liés.

Composition dans BIP. BIP (Behaviour, Interaction, Priority) est un langage conçu pour la construction de systèmes corrects, par composition de composants hétérogènes.

BIP adopte une méthodologie à trois couches correspondant au nom BIP, pour la construction des composants : une couche basse où on décrit des comportements sous la forme de réseau de Petri ou d'automate à états ; une couche intermédiaire contenant des connecteurs qui décrivent les interactions entre les comportements du niveau précédent. Enfin une couche haute, consacrée à la description d'un ensemble de règles de priorités pour l'ordonnancement des interactions.

La composition de composants BIP est binaire et consiste en la composition deux à deux des trois niveaux des différents composants. Le résultat est aussi un composant à trois niveaux. La composition est ainsi incrémentale.

Comparée à Kmelia, BIP n'est pas multi-services ; les comportements et les assertions sont définis au niveau composant dans BIP alors qu'ils le sont au niveau service dans Kmelia. Les principes de composition sont comparables au niveau des services Kmelia ; on retrouve des principes similaires (composition horizontale avec communication synchrone), les services Kmelia et les comportements des composants BIP sont modélisés par des systèmes états-transitions communicants. La composition est identique à ce niveau ; cependant dans Kmelia les connexions des services sont plus simples –canaux abstraits supports de communication– que dans BIP où ce sont des connecteurs définis par des traces d'actions. L'ordonnancement des interactions dans Kmelia est non-déterministe, il n'y a pas de priorité comme dans BIP. En revanche la composition verticale dans BIP est hiérarchique, avec des règles spécifiques de calcul du modèle d'interaction pour le composant résultant ; dans Kmelia elle se résume, dans la cas des services internes, à une composition parallèle sur un canal abstrait.

7. Conclusion et perspectives

Nous avons présenté dans cet article les principes de la composition dans l'approche Kmelia pour la conception et le développement à base de composants. La présentation est faite sous la forme d'une synthèse des travaux et résultats autour de Kmelia. Notamment nous avons distingué la composition verticale pour une structuration en profondeur des services et composants, puis la composition horizontale pour une structuration en largeur des composants. A partir de la composition nous avons montré les formes d'interactions admises dans Kmelia. Elles sont basées sur des communications via des canaux abstraits à la manière de CSP de Hoare. La dernière partie de l'article est consacrée à l'outillage de la proposition ; nous avons donné les principales caractéristiques de la plateforme (COSTO) qui accompagne Kmelia.

Il nous reste encore beaucoup de travail à faire autour de l'approche Kmelia. Parmi les pistes en cours d'exploration, nous avons la correction fonctionnelle des services (que nous explorons en utilisant la méthode B) et le raffinement de spécifications Kmelia vers des applications opérationnelles en nous appuyant sur des plateformes spécifiques (*middleware*) offrant les possibilités d'interaction necessaires.

8. Bibliographie

Abrial J.-R., *The B-Book Assigning Programs to Meanings*, Cambridge University Press, 1996. ISBN 0-521-49619-5.

Allen R., Garlan D., « A Formal Basis for Architectural Connection », *ACM Transactions on Software Engineering and Methodology*, vol. 6, n° 3, p. 213-249, July, 1997.

André P., Ardourel G., Attiogbé C., « Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition », *1ère Conférence Francophone sur les Architectures Logicielles*, Hermès, Lavoisier, p. 101-118, 2006a.

André P., Ardourel G., Attiogbé C., « Vérification d'assemblage de composants logiciels Expérimentations avec MEC », *in* M. Gourgand, F. Riane (eds), *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006*, Lavoisier, Rabat, Maroc, p. 497-506, April, 2006b.

André P., Ardourel G., Attiogbé C., « Protocoles d'utilisation de composants, spécification et analyse en Kmelia », *13e Conférence Francophone sur les Langages et Modèles à Objets*, Hermès, Lavoisier, p. 19-34, 2007.

André P., Ardourel G., Attiogbé C., Lanoix A., « Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies », *6th International Workshop on Formal Aspects of Component Software(FACS 2009)*, LNCS, 2009. to be published.

André P., Ardourel G., Attiogbé C., « A Formal Analysis Toolbox for the Kmelia Component Model », *Proceedings of ProVeCS'07 (TOOLS Europe)*, n° 567 in *Technichal Report*, ETH Zurich, 2007a.

André P., Ardourel G., Attiogbé C., « Adaptation for Hierarchical Components and Services », *Electron. Notes Theor. Comput. Sci.*, vol. 189, p. 5-20, 2007b.

André P., Ardourel G., Attiogbé C., « Defining Component Protocols with Service Composition : Illustration with the Kmelia Model », *6th International Symposium on Software Composition, SC'07*, vol. 4829 of *LNCS*, Springer, 2007c.

André P., Ardourel G., Attiogbé C., « Composing Components with Shared Services in the Kmelia Model », *7th International Symposium on Software Composition, SC'08*, vol. 4954 of *LNCS*, Springer, 2008.

Attiogbé C., André P., Ardourel G., « Checking Component Composability », *5th International Symposium on Software Composition, SC'06*, vol. 4089 of *LNCS*, Springer, 2006.

Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J.-B., « The Fractal Component Model and Its Support in Java », *Software Practice and Experience*, 2006.

Bures T., Hnetynka P., Plasil F., « SOFA 2.0 : Balancing Advanced Features in a Hierarchical Component Model », *SERA '06 : Fourth IC on Software Engineering Research, Management and Applications*, IEEE Computer Society, p. 40-48, 2006.

Chen Z., Liu Z., Stolz V., Yang L., Ravn A. P., « A Refinement Driven Component-Based Design », *12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*, IEEE Computer Society, p. 277-289, July, 2007.

Gossler G., Sifakis J., « Composition for component-based modeling », *Sci. Comput. Program.*, vol. 55, n° 1-3, p. 161-183, 2005.

Liu Z., Morisset C., Stolz V., rCOS : Theory and Tools for Componnent-based Model Driven Development, Technical Report n° 406, UNU-IIST, February, 2009. Keynote to appear in Proc. 3rd International Symposium on Fundamentals of Software Engineering, FSEN 2009, Lecture Notes in Computer Science.

Medvidovic N., Taylor R. N., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, n° 1, p. 70-93, january, 2000.

Rausch A., Reussner R., Mirandola R., Plasil F. (eds), *The Common Component Modeling Example : Comparing Software Component Models*, vol. 5153 of *LNCS*, Springer, Heidelberg, 2008.

Shaw M., Garlan D., *Software Architecture : Perspective on an Emerging Discipline*, Prentice Hall, 1996.

Sifakis J., « A Framework for Component-based Construction Extended Abstract », *SEFM'05 : Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, IEEE Computer Society, Washington, DC, USA, p. 293-300, 2005.