Correction d'assemblages de composants impliquant des interfaces paramétrées

Pascal André*, Christian Attiogbé*, Mohamed Messabihi*

*LINA - UMR CNRS 6241
2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France
(Pascal.Andre,Christian.Attiogbe,Mohamed.Messabihi)@univ-nantes.fr
http://www.lina.sciences.univ-nantes.fr/coloss/

Résumé. La démarche de construction du logiciel en partant de l'architecture, nécessite la prise en compte de la correction à différentes étapes afin d'assurer la qualité du logiciel final. Ainsi la correction est une préoccupation qui doit être prise en compte au niveau des composants et de leurs assemblages pour élaborer l'architecture logicielle. Kmelia est un langage et un modèle à composants multi-service où les composants sont abstraits et formels de façon à pouvoir y exprimer des propriétés et les vérifier. Les services de Kmelia peuvent êtres paramétrés par des données et sont dotés d'assertions (pré/post-conditions opérant sur les données). Dans cet article nous nous intéressons à la correction des modèles à composants en couvrant différents aspects: la correction au niveau des services et la correction des assemblages du point de vue des données présentes dans les interfaces des services. Nous présentons les enrichissements du langage de données de Kmelia permettant de traiter la correction au niveau des services et de l'architecture. Nous illustrons l'étude par un exemple.

1 Introduction

Les architectures logicielles présentent le grand intérêt de permettre le raisonnement sur des systèmes logiciels complexes à un niveau abstrait, c'est-à-dire en faisant abstraction des détails de conception ou d'implantation. Il est crucial de pouvoir démontrer des propriétés générales (sûreté, vivacité) de ces systèmes, ce qui est très difficile avec le code exécutable. Les propriétés peuvent concerner des composants assemblés dans les architectures ou bien les assemblages eux-mêmes. Elles couvrent aussi bien les aspects données, dynamiques ou structurels. Nous considérons des architectures logicielles à composants Allen (1997); Clements (1996); Medvidovic et Taylor (2000); Oussalah et al. (2005) décrites par des ADLs (*Architecture Description Language*) qui couvrent les aspects structurels, de données (fonctionnels) et dynamiques. Au niveau abstrait, l'architecture est perçue comme une collection de composants assemblés via des connecteurs dans des configurations Allen (1997). Les composants offrent et requièrent des services via leurs interfaces. La connexion entre composants se fait selon les modèles par des liaisons d'interfaces, de ports ou directement de services.

Notre travail porte sur la formalisation des architectures permettant la vérification de propriétés et le raffinement. Il se situe donc dans la lignée des approches formelles pour la des-

cription abstraite d'architectures logicielles. Dans des travaux précédents, nous avons introduit un modèle et un langage appelé Kmelia pour décrire formellement des assemblages et vérifier des propriétés structurelles et dynamiques Attiogbé et al. (2006). Nous avons aussi abordé dans André et al. (2006) le problème de la méthodologie de modélisation de ces architectures avec des services hiérarchiques.

Dans cet article, nous nous intéressons au langage de données et à la vérification des propriétés associées. Cet aspect couvre la définition de types de données, les expressions, les assertions, les communications, le typage des composants et des assemblages. Les propriétés d'intérêt sont la sûreté de fonctionnnement des services et la préservation des propriétés des services dans les assemblages. Les principales contributions de ce travail sont : *i)* un langage formel pour les données et assertions qui fait de Kmelia un langage riche pour la spécification d'architectures de composants, *ii)* des techniques de vérification associées.

La suite de l'article est organisée de la façon suivante : la section 2 présente les principales caractéristiques du modèle Kmelia, complété par une partie données décrite dans la section 3. L'ensemble est illustré sur un cas concret extrait du référentiel CoCoME Rausch et al. (2008) dans la section 4. Dans la section 5 nous traitons la démarche de vérification de propriétés autour de Kmelia et nous illustrons par un petit exemple. La section 6 situe notre approche parmi des travaux similaires. Enfin nous évaluons le travail et indiquons des perspectives dans la section 7.

2 Kmelia : un modèle à composants multi-services

Kmelia est un modèle de spécification de composants basés sur des descriptions de services complexes Attiogbé et al. (2006). Les composants sont *abstraits*, indépendants de leur environnement et par conséquent non exécutables. Kmelia sert à modéliser des *architectures logicielles* et leur *propriétés*. Ces modèles peuvent ensuite être raffinés vers des plate-formes d'exécution. Kmelia sert aussi de modèle commun pour l'étude de propriétés de modèles à composants et services (abstraction, interopérabilité, composabilité). Les caractéristiques principales du modèle Kmelia sont : les composants, les services et les assemblages.

Un **composant** est défini par un espace d'états, des services et une interface I. 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* I_p (resp. *requis* I_r) qui réalisent (resp. déclarent les besoins) des fonctionnalités. Les **services** sont eux-mêmes constitués d'une interface, d'une description d'état et d'assertions (pre-post conditions). Formellement, un service s est défini par un couple $(I_s,\ \mathcal{B}_s)$ où I_s est l'interface du service et $\mathcal{B}_s$ est un éventuel comportement dynamique.

L'interface du service est une abstraction des relations de composition de services, soit horizontale (dépendance) soit verticale (inclusion). La syntaxe et l'utilisation sont décrites dans André et al. (2007). Les services appelables au sein d'un autre service sont nommés sous-services et sont déclarés dans l'interface du service I_s. Formellement, I_s, l'**interface d'un service** s d'un composant C est spécifiée par un 5-uplet $\langle \sigma,\ P,\ Q,\ V_s, S_s \rangle$ où σ est la signature, P la pré-condition d'appel, Q la post-condition de déroulement, V_s un ensemble de déclarations de variables locales au service et $S_s = \langle sub_s,\ cal_s,\ req_s,\ int_s \rangle$ un quadruplet d'ensembles finis et disjoints de noms de services tels que $req_s \subseteq I_r$ et sub_s (resp. cal_s,

req_s, int_s) est l'ensemble des services offerts (resp. les requis de l'appelant, les requis d'un composant quelconque, les offerts en interne) dans le cadre du service s.

Le comportement (dynamique) d'un service est caractérisé par un automate, qui précise les enchaînements d'actions autorisés. Ces actions sont des calculs, des communications (émissions, réceptions de messages), des invocations ou retours de services. Formellement, $\mathcal{B}_s$, le **comportement d'un service** s est un système de transitions étiquetées étendu (ou *eLTS*) spécifié par un sexuplet $\langle S, L, \delta, \Phi, S_0, S_F \rangle$ où S est l'ensemble des états de $\mathcal{B}_s$, $S_0 \in S$ est l'état initial, $S_F \subset S$ est l'ensemble non vide des états finaux (le service se termine toujours), L est l'ensemble des étiquettes des transitions entre les éléments de S. $\delta : S * L \rightarrow S$ est la relation de transition entre les états de S selon les étiquettes. $\Phi : S \leftrightarrow sub_s$ est la relation d'étiquetage des états par des points d'expansion de type optionnel.

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 d'autres composants. Ces liaisons, appelées **liens d'assemblage**, établissent des canaux implicites pour les communications entre services. Les canaux sont point-à-point dans le modèle de base mais 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, elle est liée à la composition verticale des services. Une **composition** est un assemblage encapsulé dans un composant. La continuité des services est mise en œuvre par des **liens de promotion** qui servent à la promotion des services d'un composant vers ceux d'un composite (traits doubles dans la figure 1).

La hiérarchisation des services et des composants est une des caractéristiques de Kmelia qui permet une bonne lisibilité, de la flexibilité et une bonne traçabilité dans la conception des architectures André et al. (2006). Ce modèle de base a été enrichi d'une couche **protocole** André et al. (2007) permettant de définir des enchaînements licites de services. Une extension aux services **partagés** (canaux multipoints) et communication multiple est proposée dans André et al. (2008). La spécialisation se fait dans les services et non les liens.

3 Le langage de données de Kmelia

On souhaite formaliser des données et des assertions. Les **données** concernent les états de composants ou de services, les paramètres des interactions. Les **assertions** couvrent les invariants, les pré/post-conditions, les propriétés spécifiques et les prédicats. Ces assertions sont nécessaires pour vérifier des propriétés prescrites. L'expressivité des assertions permet de s'engager à vérifier statiquement aussi bien (i) la correction au niveau du services (propriétés fonctionnelles) que (ii) la correction au niveau de l'architecture (propriétés d'assemblage). Pour mettre en œuvre cet aspect **langage de données** [1], qui n'est pas nouveau dans les modèles composants, nous avons du établir un compromis entre l'expressivité (souhaitée) du langage et les contraintes liées notamment à l'intégration des aspects dynamiques des eLTS et des communications, en particulier pour la vérification cohérente de propriétés. Dans la suite de

1. Notons que la partie calcul définit la sémantique des actions dans les composants (initialisation et actions sur les transitions,...) en réutilisant la partie donnée.

cette section nous montrons l'extension de la partie donnée comprenant des déclarations de types, des expressions arithmétiques et logiques ainsi que des prédicats.

3.1 Types

Kmelia reprend les types de base usuels $Integer, Boolean, Char, String$, et fournit à l'utilisateur un moyen de définir ses propres types en suivant les règles ci-dessous :

$$
\begin{array}{lcl}
TypeDef & ::= & TypeName \\
& | & \text{"struct" "\{" } TypeDecl\ (\text{","}\ TypeDecl)^*\ \text{"\}"} \\
& | & \text{"array" "[" } LiteralValue_1\ \text{".."}\ LiteralValue_2\ \text{"]" "of"}\ TypeName \\
& | & \text{"enum" "\{" } KmlExpr\ (\text{","}\ KmlExpr)^*\ \text{"\}"} \\
& | & \text{"range"}\ LiteralValue_1\ \text{".."}\ LiteralValue_2 \\
& | & \text{"setOf"}\ TypeName
\end{array}
$$

3.2 Expressions

Une expression est construite avec des constantes, des variables et des applications d'opérateurs arithmétiques et logiques classiques $(+, *, mod, <, >=, !=, \ldots)$. Dans la suite, chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit (C : constantes, V : variables, O : opérateurs, T : types). Les identificateurs sont composés de lettres, de chiffres et du caractère "_" suivant les règles usuelles. La troisième règle exprime qu'une expression peut être une opération préfixée (d'arité quelconque) et donc elle inclut les appels de fonctions.

$$
\begin{array}{lcl}
KmlExpr & ::= & LiteralValue \\
& | & V \mid C \\
& | & O\ \text{"("}\ KmlExpr_0, \ldots\ KmlExpr_n\ \text{")"} \\
& | & KmlExpr_1\ O\ KmlExpr_2
\end{array}
$$

3.3 Assertions

Nous avons introduit des assertions sous forme de prédicats (pré/post-conditions, invariants, gardes, propriétés, ...). Voici la syntaxe simplifiée des prédicats :

$$
\begin{array}{lcll}
Pred & ::= & Cond & \text{/* condition */} \\
& | & Prop & \text{/* proposition */} \\
& | & \text{"Not" "("}\ Pred\ \text{")"} & \\
& | & Pred\ \text{"==>"}\ Pred & \\
& | & (\texttt{"\textbackslash exists"} \mid \texttt{"\textbackslash forall"})\ Pred &
\end{array}
$$

Un **invariant** de composant est une propriété qui doit être vraie à tout instant et qui est vérifiée avant et après l'exécution de chaque service. Les invariants permettent donc de capturer le sens et les caractéristiques de validité de certaines propriétés des composants. Une **précondition** de service est également un prédicat. Elle porte sur les arguments en entrée que le

client (appelant) a pour devoir de respecter. Si la (pré)condition n'est pas respectée, le fournisseur (appelé) ne s'engage pas à exécuter correctement le service appelé. Les pré-conditions doivent être transparentes, c'est à dire qu'elles doivent être nécessaires et suffisantes pour que l'appelant puisse être servi et obtienne les garanties associées aux post-conditions. Une **post-condition** est un prédicat qui garantit les sorties que le fournisseur (appelé) s'engage à respecter si le service s'exécute normalement. Le mot-clef PRE peut être utilisé dans les post-conditions ou lors du traitement des exceptions pour faire référence à l'état dans lequel était le composant juste avant l'exécution du service.

4 Un exemple simplifié de spécification

Nous illustrons le langage Kmelia par une partie de l'étude de cas CoCoME (Common Component Modelling Example Rausch et al. (2008)). Ce *benchmark* décrit un système de vente à distance sous forme d'une collection de composants (caisse, scanner, imprimante, lecteur de carte...) interconnectés et qui interagissent. Il inclut des fonctionalités 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 administratives telles que la gestion du stock et la génération des rapports, etc.

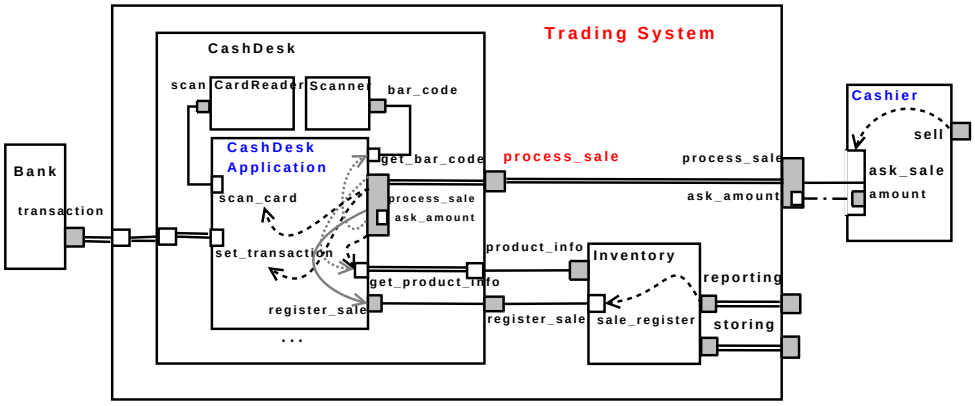


FIG. 1 – *Assemblage abstrait et simplifié du cas CoCoME.*

La figure 1 est un extrait qui se focalise sur le système de vente *Trading Sytem*, composé d'un composant de persistance *Inventory* et d'une caisse *CashDesk*. Cette dernière offre un service de vente *process_sale* en se basant sur un composant applicatif *CashDeskApplication* et des périphériques d'entrée/sortie. Les inclusions de composants dénotent la relation de composition de composants et 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). 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). Par exemple le service requis *ask_sale* est lié au service offert *process_sale*. Ce dernier fait appel à d'autres services tels que *bar_code()* vers son service appelant, *set_transaction*, *product_info*, etc. 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 1 extrait de la spécification Kmelia du composant *CashDeskApplication*. Ce service décrit le processus de vente.

Listing 1 – *Spécification partielle en Kmelia du composant :* `CashDeskApplication`

```
COMPONENT CashDeskApplication
INTERFACE
   provides : {process_sale , register_sale}
   requires : {get_product_info , set_transaction , scan_card}
TYPES
   PRODUCT : struct {price:Integer , account:Integer , total:Integer},
   PAYMENTMODE : enum {card , cash},
   IDENTIFIERS : setOf INTEGER,
   SALE_STATE : enum {open , close}
CONSTANTS
   null : Integer := -1
VARIABLES
   list_id : IDENTIFIERS,
   state : SALE_STATE
...
PROPERTIES
...
INITIALIZATION
   list_id := emptySet; state := close
...
SERVICES #--------- provided services ---------
provided process_sale(id : Integer)
Interface
   calrequires : {bar_code, amount}
   extrequires : {get_product_info , set_transaction , scan_card}
   intrequires : {register_sale}
Pre
   ((state = open) && (id \in list_id))
Variables
   prod_id : Integer , total : Integer , amount_var : Integer , rest : Integer ,
   authorisation : Boolean , credit_info : String , prod_info : PRODUCT,
   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 -- __CALLER?payment(payment_mode) --> e6,
   e6 -- [payment_mode = cash] display ("cash payment")--> e7,
   e6 -- [payment_mode = card] display("card payment") --> e13,
   e7 -- __CALLER!!amount(total) --> e8,
   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(compute_sale_string) --> e12,#()
   e12 -- __CALLER!!process_sale(true) --> f,
   e13 -- credit_info := _scan_card!!scan_card() --> e14,
```

```
   e14 -- _set_transaction!!set_transaction(credit_info, 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
...
#---------- required services ----------
required get_product_info(prod_id : Integer) : PRODUCT
...
required set_transaction (credit_info : String) : Boolean
...
END_SERVICES
```

Une description du service de vente *sell* modélisant le comportement du caissier figure dans le listing 2. Dans l'état *e4* la notation entre chevron indique que le sous-service *amount* est invocable dans cet état.

Listing 2 – *Spécification partielle en Kmelia du service :* `sell`

```
provided sell(id : Integer, mode : PAYMENTMODE)
Interface
   subprovides : {amount}
   extrequires : {ask_sale}
Variables # local to the service
   money : Integer,    sale_success : Boolean
Behavior
   init i       final f
{  i -- ... --> e0,
   e0 -- _ask_sale!!ask_sale(id) --> e1,
   e1 -- _ask_sale!new_code() --> e2,
   e2 -- _ask_sale!endSale()  --> e3,
   e2 -- _ask_sale!new_code() --> e2,
   e3 -- _ask_sale!payment(mode) --> e4,
   e4 <<amount()>>,      # subservice provided in state e4 only
   e4 -- [payment_mode = cash]_ask_sale?rest_amount(money) --> e4,
   e4 -- sale_success := _ask_sale??ask_sale() --> f
}
end
```

5 Vérification des propriétés impliquant des données et assertions

Les composants et leur assemblage peuvent être analysés sous diverses facettes. Dans un article précédent Attiogbé et al. (2006) nous avons traité l'interopérabilité statique dans les assemblages en quatre niveaux (signature, interface hiérarchique, assertions, interactions) sans toutefois détailler les assertions. Dans cet article nous nous préoccupons uniquement des propriétés fonctionnelles et notamment de la correction fonctionnelle des services à travers la vérification des assertions.

Proposition 5.1 *Les propriétés fonctionnelles exprimées par les assertions d'un service $serv$ sont préservées si et seulement si :*
(i) elles sont vérifiées localement (à l'échelle du composant offrant $serv$) soit

1. *correction des pré/post avec l'automate y compris les appels de service,*
2. *correction des pré/post vis-à-vis de l'invariant du composant,*
3. *correction des enchaînements de services inclus ;*

(ii) elles sont vérifiées globalement au niveau de l'assemblage (architecture) :

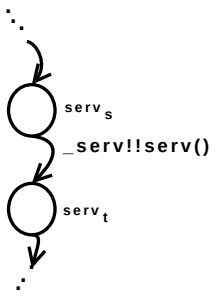
1. *compatibilité des assertions sur des liens d'assemblage,*
2. *compatibilité des assertions sur des liens de promotion.*

Vérifications locales

La vérification (i.2) se fait en utilisant des techniques classiques telles que celles utilisées dans Z ou B. La vérification (i.3) des protocoles est étudiée dans André et al. (2007). La vérification (i.1) est complexe car elle induit la construction de post-conditions à partir d'expressions Kmelia et d'automates. Nous abordons ici le problème des appels de services.

Nous rappelons que le comportement d'un service est décrit par un $eLTS$. Les *propriétés* d'un état s (notées $state\_properties(s)$) dans un $eLTS$ de service sont les propriétés souhaitées (exprimées par le spécifieur, ou générées par un transformateur de prédicats) à cet état.

Considérons dans un $eLTS$ une transition étiquetée par un appel de service $serv$; nous appelons l'état source de cette transition $serv_s$ et $serv_t$ son état de destination.



Notre hypothèse [2] pour la vérification des assertions du service $serv$ est de ne tenir compte que des variables qui sont présentes dans $state\_properties(serv_s)$ et/ou $state\_properties(serv_t)$. La vérification dans ce cas est définie par l'algorithme ($check_c$) ci-dessous :

$$check_c(serv) = \begin{cases} state\_properties(serv_s) \vdash P_{serv} \\ Q_{serv} \vdash state\_properties(serv_t) \end{cases}$$

Intuitivement, l'algorithme $check_c$ garantit que $state\_properties(serv_s)$ n'est pas contradictoire avec P_{serv}, et de la même manière que Q_{serv} ne contredit pas $state\_properties(serv_t)$.

Vérification au niveau des assemblages

Considérons un service $serv$ avec les assertions P_{serv} et Q_{serv}. Un service requis (resp. offert) est noté $serv_r$ (resp. $serv_p$) .

2. Notons que cette hypothèse est relachée si les services utilisent un alphabet commun.

Pour un lien d'assemblage, on vérifie les assertions de type contrat. L'algorithme de vérification d'un lien d'assemblage renvoie *vrai* lorsque (P_{serv_r} est plus forte que P_{serv_p} et Q_{serv_p} est plus forte que Q_{serv_r}. Nous formalisons cet algorithme ($check_a$) de la façon suivante

$$check_a(link(serv_r, serv_p)) \Rightarrow (P_{serv_r} \Rightarrow P_{serv_p}) \wedge (Q_{serv_p} \Rightarrow Q_{serv_r})$$

Dans un lien de promotion d'un service (offert ou requis) on véhicule toutes les propriétés du service promu $serv$ au service promouvant (noté p_serv) et éventuellement on en ajoute d'autres. L'algorithme de vérification d'un lien de promotion de service requis renvoie *vrai* lorsque

$$check_p(promote(serv_r, p\_serv_r)) \Rightarrow (P_{serv_r} \Rightarrow P_{p\_serv_r}) \wedge (Q_{p\_serv_r} \Rightarrow Q_{serv_r})$$

L'algorithme de vérification d'un lien de promotion de service offert renvoie *vrai* lorsque

$$check_p(promote(serv_p, p\_serv_p)) \Rightarrow (P_{p\_serv_p} \Rightarrow P_{serv_p}) \wedge (Q_{serv_p} \Rightarrow Q_{p\_serv_p})$$

Notons que l'algorithme $check_p()$ peut être appelé récursivement pour vérifier les assertions des sous-services. Ces algorithmes peuvent être implantés avec B en s'inspirant des travaux sur la compatibilité d'interfaces décrites en B Lanoix et al. (2008).

Illustration

Illustrons la vérification (locale) d'appel de service sur l'exemple CoCoME. Nous nous intéressons au service `sell()` du composant `Cashier`. Lors de l'appel du service `ask_sale(id)` le service `sell()` envoie son identifiant `id` comme paramètre. Selon notre démarche de vérification, nous avons une obligation de preuve *op1* : $(id \in list\_id)$. Elle est issue de la précondition du service `process_sale()` et exprime que l'`id` doit appartenir à la liste des identifiants dans `CashDeskApplication`. En utilisant la procédure de vérification $check_c$, on vérifie non seulement la compatibilité des types mais on prouve également l'obligation de preuve *op1*. Cette obligation de preuve aurait permis de détecter une erreur si, par exemple, le service `ask-sale(id)` était appelé avec un identifiant inconnu.

6 Travaux connexes

La prise en compte des données dans les modèles à composants n'est pas nouvelles par contre il est plus original de combiner données, contrats, dynamique et communications dans un langage intégré. Kmelia le fait à un niveau qui permet la vérification *statique* de propriétés relatives aux aspects structurels, dynamiques et fonctionnels. En particulier le langage de données doit servir non pas uniquement à la génération de code mais à la vérification des modèles abstraits.

Les modèles opérationnels tels que Corba, EJB ou .NET ne permettent pas de raisonner au niveau architectural qui nous intéresse ici. Il en est de même pour les modèles proches de la programmation tels que Java/A Baumeister et al. (2006) ou ArchJava Aldrich et al. (2002). Certains modèles Bruneton et al. (2006); Canal et al. (2003) proposent de repousser le langage de données au niveau de l'implantation : les types de données et les calculs associés sont alors définis, implantés et vérifiés par le compilateur. Ce choix pose problème pour les architectures

logicielles car il est trop tardif, de plus il s'intègre mal avec les vérifications de structure, de contrat et de dynamique. D'autres modèles Collet et Rousseau (2005); Schmidt (2003) prennent en compte des types de données et des contrats mais pas d'aspects dynamiques.

Les modèles avancés sur les aspects dynamiques Canal et al. (2003); Yellin et Strom (1997); Magee et al. (1999); Allen (1997); Bures et al. (2006) Dans Wright par exemple, la partie comportementale basée sur CSP est très détaillée (spécification et vérification) tandis que la partie donnée est mineure « *We will not carry out any specific formal proof using the developed model.* » Allen (1997) . Un modèle d'état et des opérations sont décrits dans un sous-ensemble de Z ; une opération correspond à un événement dans le modèle comportemental. Dans Pavel et al. (2005) la prise en compte des aspects données se fait sous forme de spécifications algébrique qui s'intègrent bien dans une vérification symbolique avec des systèmes de transition. Cependant le modèle ne supporte pas d'assertions.

Certaines approches se focalisent sur les contrats dans les communications. Par exemple, dans Cariou (2003) l'idée est de définir des abstractions de communications (sortes de collaborations à la UML) puis de les réifier par des *medium* ou composants de communication. La partie contrat est décrite en OCL. C'est une approche complémentaire de la nôtre puisque dans Kmelia nous faisons abstraction de la mise en œuvre de la communication. Dans Carrez et al. (2003) l'idée est d'associer des contraintes (*may/must*) aux interactions définies dans les interfaces, et ainsi de définir des contrats comportementaux liant le client et le serveur. En Kmelia, la distinction entre contrainte du fournisseur et du demandeur se fait d'un point de vue méthodologique et non syntaxique. Par ailleurs, un service est atomique dans le modèle de Carrez (une opération avec son contrat) alors que dans Kmelia il est hiérarchique et son comportement dynamique (eLTS) doit respecter les assertions.

Fractal Coupaye et Stefani (2006) propose différentes approches basées sur la séparation des préoccupations. L'aspect structurel est pris en compte dans Fractal ADL Coupaye et al. (2007) ; les assertions sont traitées dans ConFract Collet et Rousseau (2005) et enfin la dynamique est étudiée dans Vercors Barros et al. (2007) ou Fractal/SOFA Bureš et al. (2008).

Kmelia présente l'avantage de mettre en avant la notion de service, ce qui procure un pont relativement naturel avec les architectures à services.

7 Conclusion et perspectives

Nous avons présenté des enrichissements du langage Kmelia notamment sa partie données par des assertions de la forme pré/post. La vérification syntaxique est opérationnelle dans l'outil COSTO. En fonction de cela nous avons défini une procédure de vérification des propriétés des services (propriétés exprimées sur les états de l'automate de comportement) et leur préservation au niveau des assemblages. Elle doit s'intégrer avec les vérifications de propriétés sur les aspects structurels et dynamiques.

A partir de cette procédure de vérification, nous avons les obligations à vérifier afin d'assurer la correction des assemblages de composants impliquant des services avec des interfaces dotés de pré et post-conditions. Les vérifications sont actuellement manuelles. Les expérimentations avec le langage B semblent intéressantes de ce point de vue. Il nous reste à mettre en œuvre à l'aide d'outils ces vérifications sur des cas d'étude plus nombreux afin de les améliorer et les rendre systématiques.

Références

Aldrich, J., C. Chambers, et D. Notkin (2002). Archjava : connecting software architecture to implementation. In *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, pp. 187–197. ACM.

Allen, R. (1997). *A Formal Approach to Software Architecture*. Ph. D. thesis, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

André, P., G. Ardourel, et C. Attiogbé (2006). Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition. In *1ère Conférence Francophone sur les Architectures Logicielles*, pp. 101–118. Hermès, Lavoisier.

André, P., G. Ardourel, et C. Attiogbé (2007). Defining Component Protocols with Service Composition : Illustration with the Kmelia Model. In *6th International Symposium on Software Composition, SC'07*, Volume 4829 of *LNCS*. Springer.

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

Attiogbé, C., P. André, et G. Ardourel (2006). Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, Volume 4089 of *LNCS*. Springer.

Barros, T., A. Cansado, E. Madelaine, et M. Rivera (2007). Model-checking distributed components : The vercors platform. *Electron. Notes Theor. Comput. Sci. 182*, 3–16.

Baumeister, H., F. Hacklinger, R. Hennicker, A. Knapp, et M. Wirsing (2006). A component model for architectural programming. *Electr. Notes Theor. Comput. Sci. 160*, 75–96.

Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma, et J.-B. Stefani (2006). The Fractal Component Model and Its Support in Java. *Software Practice and Experience 36*(11-12).

Bureš, T., M. Děcký, P. Hnětynka, J. Kofroň, P. Parízek, F. Plášil, T. Poch, O. Šerý, et P. Tůma (2008). *CoCoME in SOFA*, pp. 1–2. Volume 5153 of Rausch et al. Rausch et al. (2008).

Bures, T., P. Hnetynka, et F. Plasil (2006). Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *SERA '06 : Fourth IC on Software Engineering Research, Management and Applications*, pp. 40–48. IEEE Computer Society.

Canal, C., L. Fuentes, E. Pimentel, J. M. Troya, et A. Vallecillo (2003). Adding Roles to CORBA Objects. *IEEE Trans. Softw. Eng. 29*(3), 242–260.

Cariou, E. (2003). *Contribution à un processus de réification d'abstractions de communication*. Ph. D. thesis, Université de Rennes 1.

Carrez, C., A. Fantechi, et E. Najm (2003). Contrats comportementaux pour un assemblage sain de composants. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2003)*, Paris, France.

Clements, P. C. (1996). A survey of architecture description languages. In *IWSSD '96 : Proceedings of the 8th International Workshop on Software Specification and Design*, Washington, DC, USA, pp. 16. IEEE Computer Society.

Collet, P. et R. Rousseau (2005). ConFract : un système pour contractualiser des composants logiciels hiérarchiques. *L'Objet, LMO'05 11*(1-2), 223–238.

Coupaye, T., V. Quéma, L. Seinturier, et J.-B. Stefani (2007). *Intergiciel et Construc-*

tion d'Applications Réparties, Chapter Le système de composants Fractal. InriAlpes. sardes.inrialpes.fr/ecole/livre/pub/.

Coupaye, T. et J.-B. Stefani (2006). Fractal component-based software engineering. In M. Südholt et C. Consel (Eds.), *ECOOP Workshops*, Volume 4379 of *Lecture Notes in Computer Science*, pp. 117–129. Springer.

Lanoix, A., S. Colin, et J. Souquières (2008). Développement formel par composants : assemblage et vérification à l'aide de B. *Technique et Science Informatiques (TSI) 26*(8), 1007–1032. Numéro spécial AFADL07.

Magee, J., J. Kramer, et D. Giannakopoulou (1999). Behaviour analysis of software architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, Deventer, The Netherlands, pp. 35–50. Kluwer, B.V.

Medvidovic, N. et R. N. Taylor (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering 26*(1), 70–93.

Oussalah, M., T. Khammaci, et A. Smeda (2005). *Les composants : définitions et concepts de base*, Chapter 1, pp. 1–18. Les systèmes à base de composants : principes et fondements. M. Oussalah et al. Eds, Editions Vuibert.

Pavel, S., J. Noye, P. Poizat, et J.-C. Royer (2005). Java Implementation of a Component Model with Explicit Symbolic Protocols. In *4th International Symposium on Software Composition, SC'05*, Volume 3628 of *LNCS*. Springer.

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

Schmidt, H. (2003). Trustworthy components-compositionality and prediction. *J. Syst. Softw. 65*(3), 215–225.

Yellin, D. et R. Strom (1997). Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems 19*(2), 292–333.

Summary

Software development from architectures requires to take into account correctness concerns at different steps in order to ensure the quality of the developped software. Therefore the correctness concern should be taken into account at the level of components and their assemblies to build architecture. Kmelia is a language and also a multi-service component model where components are abstract and formal so that one can express and verify properties. The Kmelia services may be parameterised by data and they are equipped assertions in the form of pre/post-conditions. In this article we focus on the correctness of component models by covering various aspects: service levelcorrectness and assembly level correctness from the point of view of the data present in the service interfaces. We present the enrichment of the Kmelia data language that enables one to deal with correctness at the service and architecture levels. The work is illustrated by an example.