Preuve de cohérence de composants Kmelia à l'aide de la méthode B

Mohamed Messabihi, Pascal André, Christian Attiogbé

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

Résumé. Pour répondre aux objectifs de qualité dans la construction de systèmes à base de composants logiciels et améliorer la confiance dans les composants et leur assemblage, il est nécessaire de disposer d'un support de correction. Kmelia est un modèle à composants multi-services dans lequel les composants sont abstraits et formels de façon à pouvoir y exprimer des propriétés et les vérifier. Dans cet article nous étudions l'automatisation de la vérification de cohérence des composants Kmelia et de leur assemblage en nous servant de la méthode B. Pour ce faire, des machines B sont extraites des composants Kmelia et vérifiées en utilisant l'Atelier B, établissant de fait les propriétés au niveau de Kmelia. La démarche est illustrée par un exemple de gestion de stock.

1 Introduction

Pour répondre aux objectifs de qualité dans la construction de systèmes à base de composants logiciels et améliorer la confiance dans les composants et leur assemblage, disposer d'un support de correction est nécessaire Meyer (2003). Ce support est basé sur des modèles formels et des outils de vérification de propriétés générales (sûreté, vivacité). Il est plus simple à mettre en œuvre au niveau des architectures que du code exécutable car elles permettent de s'abstraire des détails de conception ou d'implantation. Le travail présenté ici est applicable aux architectures logicielles à composants telles que décrites dans Allen et Garlan (1997); Clements (1996); Medvidovic et Taylor (2000); Oussalah et al. (2005). Dans ces approches et au niveau abstrait, l'architecture est perçue comme une collection de composants assemblés via des connecteurs dans des configurations. 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 modèles à composants permettant la vérification de propriétés et le raffinement. Dans Attiogbé et al. (2006), nous avons introduit un modèle et un langage appelé Kmelia pour décrire formellement des assemblages et vérifier des propriétés structurelles et dynamiques. Dans Kmelia les composants sont assemblés et composés par des liens entre les services de leurs interfaces. Dans André et al. (2009b) nous avons introduit un langage de données pour la spécification des types de données, des actions et des assertions (invariant, pré/post-conditions). Nous avions esquissé les vérifications formelles associées. Dans

cet article, nous nous intéressons à l'automatisation de la vérification des propriétés de préservation de cohérence des composants par les services et de respect des contrats d'assemblage. Les principales contributions de ce travail sont : *i)* un cadre pour la vérification de cohérence des composants et des contrats d'assemblage, *ii)* des techniques de vérification par transformation en B, *iii)* un outil qui automatise les preuves des propriétés attendues.

La suite de l'article est organisée de la façon suivante : la section 2 rappelle les principales caractéristiques du modèle Kmelia pour les propriétés visées. L'ensemble est illustré sur un cas concret de gestion de stock. La section 3 donne un panorama des vérifications à faire pour des composants Kmelia. La section 4 est une introduction à la méthode B. Dans la section 5 nous détaillons la démarche de vérification qui fait l'objet de l'article et son automatisation. La section 6 illustre l'automatisation des vérifications de la section 3. La section 7 situe notre approche parmi des travaux similaires. Enfin nous évaluons le travail et indiquons des perspectives dans la section 8. Le lecteur trouvera de détail des exemples sur le site web `http://www.lina.sciences.univ-nantes.fr/coloss/download/cal10_a.zip`.

2 Kmelia : un modèle à composants multi-services

Kmelia est un modèle et un langage de spécification de composants multi-services initié par Attiogbé et al. (2006). Il sert à modéliser des *assemblages de composants* et leur *propriétés*. Les composants sont *abstraits*, indépendants de leur environnement et par conséquent non exécutables. 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 de ce modèle sont : les composants, les services, les assemblages, les composites. Une description formelle en est proposée dans Attiogbé et al. (2006); André et al. (2009a). Nous les illustrons ici à travers un exemple simplifié de gestion de stock. *Un processus de vente* vending *permet de gérer les références dans un catalogue (*catalog*) et les quantités stockées (*stock*). Les administrateurs peuvent ajouter ou retirer des références selon les règles de gestion établies telles que : on ne peut ajouter une référence que si elle ne figure pas déjà dans le catalogue ou on ne peut retirer une référence que si son stock est vide.*

Une architecture logicielle en Kmelia est définie par un assemblage de composants. Ces composants encapsulent des services dans des interfaces, qui distinguent les *services offerts*, réalisant une fonctionnalité, des *services requis*, déclarant les besoins de fonctionnalités. Les composants sont assemblés via les services de leur interface par des **liens d'assemblage** qui établissent des canaux implicites de communication entre services. Un **composite** est un composant qui encapsule un assemblage. La continuité des services est mise en œuvre par des **liens de promotion** des services d'un composant vers ceux d'un composite. La figure 1 montre l'assemblage partiel d'une application de gestion de stock en se focalisant sur le processus de vente (service vending). Ce service requiert quatre services, dont le service requis addItem qui est lié au service offert newReference par un lien d'assemblage. Un *sous-lien* est un lien dont le contexte est défini dans un autre lien. Par exemple, le service code est offert par addItem dans le cadre d'un ajout de référence newReference.

Un **composant** est caractérisé principalement par une interface, un espace d'états et des services. L'interface déclare les *services offerts* et *requis* d'un composant. L'espace d'état est un ensemble de constantes et de variables typées, contraintes par un invariant et une initiali-

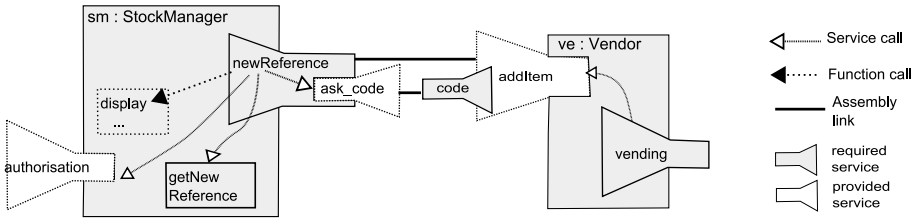

FIG. 1 – *Assemblage simplifié de la gestion de stock*

sation. On appelle **espace d'état observable** le sous-ensemble qui est déclaré visible par le concepteur du composant. En pratique on utilise l'espace d'état observable pour définir des contrats d'assemblage ou de promotion. Dans le listing 2, seule la variable d'état catalog (qui contient les références de produits) est observable (les codes vendeurs, les libellés et les quantités en stock ne le sont pas). Les données, assertions et expressions sont décrites en utilisant le langage de données introduit dans André et al. (2009b). Il couvre les types et opérateurs usuels (entiers, booléens, énumérations, structures, tableaux et ensembles). L'utilisateur peut déclarer ses propres types dans les spécifications de composants ou dans des bibliothèques comme STOCKLIB. Les assertions sont des prédicats étiquetés par un label comme @borned dans le Listing 2.

Listing 1 – *Extrait de STOCKLIB*

```
LIBRARY_CONSTANTS
  obs maxRef: Integer:=100;
  obs noReference : Integer := 0;
  noQuantity : Integer := -1;
```

Listing 2 – *Spécification Kmelia de StockManager*

```
COMPONENT StockManager
INTERFACE
  provides : {newReference, removeReference, storeItem, orderItem}
  requires : {authorisation}
USES {STOCKLIB} // library of the project types and functions
TYPES
  Reference :: range 1..maxRef
VARIABLES
  vendorCodes : setOf Integer; //authorised administrators
  obs catalog : setOf Reference; // product id = index of the arrays
  plabels : array [Reference] of String; //product description
  pstock : array [Reference] of Integer //product quantity
INVARIANT
  obs @borned: size(catalog)<=maxRef,
  @referenced: forall ref : Reference | includes(catalog,ref) implies
     (plabels[ref] <> emptyString and pstock[ref] <> noQuantity),
  @notreferenced: forall ref : Reference | excludes(catalog,ref) implies
     (plabels[ref] = emptyString and pstock[ref] = noQuantity)
INITIALIZATION
  catalog := emptySet;
  vendorCodes := emptySet;     //filled by a required service
  plabels:= arrayInit(plabels,emptyString); //consistent with ..
  pstock := arrayInit(pstock,noQuantity);   //..empty catalog
```

Les **services** sont eux-mêmes constitués d'une interface, d'une description d'état et d'assertions (pré/post-conditions) et d'un comportement dynamique. Ce dernier n'est pas détaillé dans

cet article. Il met en évidence sous forme d'un système de transition étiqueté étendu (eLTS) les enchaînements d'actions autorisés par le service. Ces actions sont des calculs, des communications (émissions/réceptions de messages, invocations/retours de services). L'interface du service est une abstraction des relations de composition horizontale (dépendance) ou verticale (inclusion) de services. Les services appelables au sein d'un autre service sont qualifiés de sous-services et déclarés dans l'interface du service. Dans le listing 3, le service newReference requiert un service ask_code de son appelant et un service interne getNewReference. Inversement dans le listing 4 le service addItem propose un service code.

Listing 3 – *Spécification Kmelia de StockManager:newReference*

```
provided newReference () : Integer  //Result = ProductId or noReference
Interface
   calrequires : {ask_code} #required from the caller
   intrequires : {getNewReference}
Pre
   size(catalog) < maxRef #the catalog is not full
// LTS omitted here – see web appendix
Post
 @resultRange:((Result >=1 and Result <=maxRef) or (Result = noReference)),
 @resultValue:(Result <> noReference) implies (notIn(old(catalog),Result)
                  and catalog = add(old(catalog),Result)),
 @noresultValue:(Result = noReference) implies Unchanged{catalog},
 local @refAndQuantity: (Result <> noReference) implies
         (pstock[Result] = 0 and plabels[Result] <> emptyString and
         (forall i : Reference | (i <> Result) implies
            (pstock[i] = old(pstock)[i] and plabels[i] = old(plabels)[i] )) ),
 local @NorefAndQuantity: (Result = noReference) implies Unchanged{pstock,plabels}
End
```

La pré-condition indique que le catalogue ne doit pas être plein. Elle porte sur la partie observable de l'espace d'état du composant. La post-condition comprend une partie observable : le résultat (variable result) est soit une référence nulle (noReference) soit une nouvelle référence (dans l'intervalle autorisé) ; s'il s'agit d'une nouvelle référence alors que le nouveau catalogue est l'ancienne valeur (indiquée par le mot-clé old) à laquelle on a ajouté cette référence. Dans le cas d'une référence nulle, le catalogue est inchangé (mot-clé unchanged). La post-condition comprend aussi une partie non observable (ou *locale*) indiquant les effets sur l'espace d'état non-observable. Pour un service requis, on "imagine" une partie de cet espace observable, qu'on modélise sous forme d'un **espace d'état virtuel**. Dans l'exemple du listing 4, le service addItem perçoit le catalogue sous forme de deux booléens. La pré-condition "observable" porte sur cet espace virtuel, de même que la post-condition. La pré-condition "locale" ajoute des conditions liées à l'espace d'état du composant du service requis.

Listing 4 – *Spécification Kmelia de Service addItem*

```
required addItem () : Integer
Interface
  subprovides : {code}
Virtual Variables
  catalogFull : Boolean;
  catalogEmpty : Boolean     //possibly catalogSize
Virtual Invariant not(catalogEmpty and catalogFull)
Pre True // after analysis we have detected an error specification
         // we have modified the precondition by [not catalogFull]
  //No LTS
Post
  @ref: (Result <> noReference) implies (not catalogEmpty),
  @noref: (Result = noReference) implies Unchanged{catalogEmpty,catalogFull}
```

3 Vérification de propriétés des spécifications Kmelia

La correction des composants concerne en partie la cohérence des services par rapport aux propriétés du composant qui les contient ; la correction des assemblages concerne elle la cohérence des liaisons effectuées entre les services requis et offerts de différents composants. La méthode que nous avons adoptée dans ce cas est basée sur le principe suivant : un service offert est un raffinement d'un service requis (voir André et al. (2009a)). Nous nous intéressons ici à la correction des composants et assemblages vis-à-vis des assertions. La correction de la dynamique est traitée dans Attiogbé et al. (2006). Dans André et al. (2009b) nous avions établi un cadre pour la vérification de diverses propriétés liées aux données et aux actions (décrivant les comportements). Nous les détaillons ici ainsi que leur automatisation.

(PI)	*Préservation de l'invariant par les services offerts* : vérification des pré/post-conditions par rapport à l'invariant du composant.
(PO)	*Préservation de l'invariant observable par les services offerts* : variante de PI en se fondant uniquement sur l'invariant observable.
(PV)	*Préservation de l'invariant virtuel par les services requis* : vérification des pré/post-conditions par rapport à l'invariant virtuel.
(AA)	*Cohérence des actions/assertions* : vérification des pré/post-conditions par rapport au comportement dynamique.
(CA)	*Cohérence des contrats d'assemblage* : vérification des pré/post-conditions du service offert par rapport aux pré/post-conditions du service requis.
(CP)	*Cohérence des contrats de promotion* : vérification des pré/post-conditions du ser- vice du composants par rapport à la pré/post-condition du service promu du composite.

TAB. 1 – *Vérifications liées aux assertions en Kmelia*

Nous ne traitons pas ici du cas (AA) relatif à la correction du comportement dynamique d'un service par rapport à l'abstraction sous forme de pré/post-conditions. En somme les vérifications à effectuer concernent des propriétés de cohérence : cohérence des services des composants, cohérence des assemblages des composants.

Nous utilisons la méthode B pour mettre en œuvre la démarche de preuve de correction, et pour l'assistance à la preuve de cohérence. En B, on vérifie la cohérence des modèles construits par rapport à un invariant. La méthode B génère systématiquement des obligations de preuve selon la sémantique de préservation des propriétés invariantes. Ce cadre de preuve est adéquat aux preuves de cohérence pour les composants et assemblages Kmelia. Nous transposons donc la question de la vérification de cohérence de composants Kmelia en une question de vérification de cohérence en B.

Dans la suite, nous montrons comment des spécifications B sont systématiquement extraites de spécifications Kmelia. La démarche est outillée ; nous avons étendu notre plateforme logicielle COSTO [1] avec un nouveau module (Kml2B) qui réalise l'extraction des machines B, qui sont ensuite prouvées avec les outils associés à B.

1. COmponent Study TOolkit dédié au langage Kmelia

4 La méthode B : un aperçu

B est une méthode pour analyser, spécifier, concevoir et implanter des systèmes logiciels Abrial (1996). Elle permet de décrire des modèles abstraits de logiciels à l'aide de la théorie des ensembles et de la logique du premier ordre étendue avec la théorie des substitutions généralisées. A partir de ces modèles abstraits appelés machines abstraites, B propose des techniques de raffinements successifs aboutissant au code exécutable. En B, en vue de construire des modèles et des logiciels corrects, des preuves de cohérence des machines et de leurs raffinements sont effectués. La méthode B génère des obligations de preuve pour chaque niveau de raisonnement. Des outils de preuve existent pour effectuer automatiquement ou de façon interactive ces preuves. La méthode B est en pleine évolution, actuellement on peut l'utiliser sous sa version dite classique mais aussi sous la version Event-B pour la spécification de systèmes plus généraux Abrial et al. (2006); Abrial et Hallerstede (2007). Le cadre de preuves de propriétés et de raffinement successifs n'a pas fondamentalement changé mais a été étendu.

Machines abstraites B : Une machine abstraite B comporte en dehors des paramètres et de diverses clauses de structuration, les descriptions d'une partie statique et d'une partie dynamique. La partie statique spécifie un espace d'états à l'aide d'une liste d'ensembles, de définitions, de constantes, de variables et d'un invariant (Inv) qui est un prédicat exprimant les propriétés globales de tous les états du système spécifié ; l'invariant capture ainsi la sémantique voulue par le spécifieur. Ces différentes descriptions apparaissent dans des clauses comme dans les exemples que nous introduirons. La partie dynamique décrit les opérations fournies par la machine. Les opérations d'une machine B, dont nécessairement une initialisation (U), sont modélisées chacune par une pré-condition (Pre) et des substitutions généralisées S (ce sont des transformateurs de prédicats, à la *weakest-precondition* de Dijkstra).

Preuve de cohérence : La cohérence d'une machine abstraite est établie par la preuve que son initialisation établit l'invariant (noté $[Inv]U$) et la preuve que chacune des opérations appelées sous la pré-condition (Pre) préserve l'invariant : ($I \wedge Pre \Rightarrow [S]Inv$) ; ces obligations de preuve sont systématiquement générées par les outils associés à B.

Machines de raffinement : elles concrétisent des machines plus abstraites. Le raffinement concerne les données et les opérations. Chaque raffinement inclut un invariant de liaison entre les variables qu'il contient et celles de la machine abstraite qu'il raffine ; cet invariant permet d'établir et de prouver que le raffinement est correct par rapport à la machine raffinée.

Un des avantages de la méthode B, est la souplesse qu'elle offre pour modéliser et prouver des propriétés exprimées sous forme d'invariant. Nous allons exploiter cette possibilité dans la suite de notre présentation.

5 Extraction de machines B des spécifications Kmelia

Afin de vérifier systématiquement les propriétés liées aux contrats (page 5), nous proposons de construire des machines B à partir de spécifications Kmelia puis de vérifier ces machines grâce aux outils de B. Nous procédons en trois phases :

1. Pour chaque composant C décrit en Kmelia on extrait
   – une machine C dont l'espace d'état est déduit directement de celui du composant. Les services srv_i dans C sont traduits par des opérations srv_i dans la même machine.

– Une machine Cobs restreinte aux éléments observables de C est également extraite à partir du composant C en appliquant le principe ci-dessus.

Les machines B servent de support à la preuve de la propriété `(PI)` de la section 3.

2. Pour chaque lien d'assemblage `sr-sp` on extrait une machine `sr` pour le service requis et son espace d'état virtuel, et un raffinement `sr_sp_ref` du service offert `sp`. Les obligations de preuves B de ce raffinement permettent de vérifier la propriété `(CA)` de la section 3.
3. Les liens de promotions sont traduits de la même manière que les liens d'assemblages et ne seront pas traités ici.

Afin de formaliser la traduction du modèle Kmelia, nous avons introduit un morphisme $\mathbb{B}$ pour définir les règles de transformations d'un arbre de syntaxe abstraite Kmelia en un arbre de syntaxe abstraite B, préservant la sémantique des constructions Kmelia.

Types et expressions

Les types Integer, Boolean, range, enum, struct sont directement traduits en B par le morphisme $\mathbb{B}$. Le type générique setOf est fixé par l'ensemble des partie $\mathbb{P}$ et le type array est traduit par une fonction totale. Par exemple, la variable pstock du listing 2 s'écrit :

$$\begin{aligned}
\mathbb{B}\,[\text{pstock : : array [1..MaxInt] of setOf Integer}] &\overset{\mathbb{B}}{=} \text{pstock} = \mathbb{B}\,[1]..\mathbb{B}\,[\text{MaxInt}] \text{ -> } \mathbb{B}\,[\text{setOf Integer}] \\
&\overset{\mathbb{B}}{=} \text{pstock} = 1..\text{MaxInt} \rightarrow \mathbb{P}(\mathbb{B}\,[\text{Integer}]) \\
&\overset{\mathbb{B}}{=} \text{pstock} = 1..\text{MaxInt} \rightarrow \mathbb{P}(\text{INT})
\end{aligned}$$

Les expressions sont définies sur des opérateurs et des fonctions. Pour les types prédéfinis de Kmelia, la traduction prend en compte la conversion d'opérateurs Kmelia en fonctions B ou vice-versa, ainsi que la traduction des arguments. Un traitement spécifique est associé à la fonction prédéfinie old(x) (qui dans une post-condition désigne la valeur d'une variable x avant l'appel du service) basée sur la substitution ANY.

Règles sémantiques :

$$\begin{aligned}
&\mathbb{B}\,[\text{Var|Cst})] &&\overset{\mathbb{B}}{=} \text{Var}\,|\,\text{Cst} \\
&\mathbb{B}\,[\text{KmlUnaryOp (Op, KmlExpr)}] &&\overset{\mathbb{B}}{=} \text{BExpr } (\mathbb{B}\,[\text{Op}],\ \mathbb{B}[\text{KmlExpr}]) \\
&\mathbb{B}\,[\text{KmlBinOp (Op, KmlExpr, KmlExpr)}] &&\overset{\mathbb{B}}{=} \text{BExpr } (\mathbb{B}\,[\text{Op}],\ \mathbb{B}[\text{KmlExpr}],\ \mathbb{B}\,[\text{KmlExpr}]) \\
&\mathbb{B}\,[\text{KmlPred (Quantifier, ListExpr, KmlExpr)}] &&\overset{\mathbb{B}}{=} \text{BPred } (\mathbb{B}\,[\text{Quantifier}],\ \mathbb{B}\,[\text{ListExpr}],\ \mathbb{B}\,[\text{KmlExpr}]) \\
&\mathbb{B}\,[\text{KmlFunCall (fname, ListExpr)}] &&\overset{\mathbb{B}}{=} \text{BExpr } (\text{fname},\ \mathbb{B}\,[\text{ListExpr}]) \\
&\mathbb{B}\,[\text{KmlSetExpr (OP, KmlExpr)}] &&\overset{\mathbb{B}}{=} \mathbb{B}\,[\text{KmlUnaryOp (Op, KmlExpr)}] \\
&\mathbb{B}\,[\text{KmlSetExpr (OP, KmlExpr, KmlExpr)}] &&\overset{\mathbb{B}}{=} \mathbb{B}\,[\text{KmlBinOp (Op, KmlExpr, KmlExpr)}]
\end{aligned}$$

Exemple du prédicat @resultValue du listing 2 :

$$\begin{aligned}
&\mathbb{B}\,[(\text{Result} <> \text{noReference}) \text{ implies } (\text{notIn}(\text{catalog},\text{Result}))] \\
&\quad\overset{\mathbb{B}}{=} \mathbb{B}\,[\text{KmlBinOp ('implies', [KmlBinOp ('<>', Result, noReference)],} \\
&\qquad\quad \text{[KmlFunCall ('notIn', ListExpr (catalog,Result))]]} \\
&\quad\overset{\mathbb{B}}{=} \text{BExpr('}\Rightarrow\text{'},}\ \mathbb{B}\,[\text{Result} <> \text{noReference}],\ \mathbb{B}\,[\text{notIn}(\text{catalog},\text{Result})]) \\
&\quad\overset{\mathbb{B}}{=} (\text{BExpr('}\neq\text{'},}\ \mathbb{B}\,[\text{Result}],\ \mathbb{B}\,[\text{noReference}])) \Rightarrow (\text{BExpr('}\notin\text{'},}\ \mathbb{B}\,[\text{Result}],\ \mathbb{B}\,[\text{catalog}])) \\
&\quad\overset{\mathbb{B}}{=} (\,\text{Result} \neq \text{noReference}) \Rightarrow (\,\text{Result} \notin \text{catalog}\,)
\end{aligned}$$

Composants et services

L'espace d'état $\mathcal{W} = \langle T, V, type, Inv, Init\rangle$ d'un composant C (où $V = V_c \cup V_v$ constantes et variables typées dans T) est traduit par $\mathbb{B}[\mathcal{W}] \stackrel{\mathbb{B}}{=} \langle \mathbb{B}[T], \mathbb{B}[V], \mathbb{B}[type], \mathbb{B}[Inv], \mathbb{B}[Init]\rangle$.

$$\mathbb{B}[\mathcal{W} = \langle T, V, type, Inv\rangle] \stackrel{\mathbb{B}}{=} \begin{cases} \text{MACHINE} & C \\ \text{SETS} & \mathbb{B}[T] \\ \text{CONSTANTES} & \mathbb{B}[V_c] \\ \text{PROPERTIES} & \mathbb{B}[type_c] \\ \text{VARIABLES} & \mathbb{B}[V_v] \\ \text{INVARIANT} & \mathbb{B}[type_v] \wedge \mathbb{B}[Inv] \\ \text{INITIALISATION} & \mathbb{B}[Init] \\ \text{OPERATIONS} & \mathbb{B}[\mathcal{D}^P] \end{cases}$$

où :

- $\mathbb{B}[T] \stackrel{\mathbb{B}}{=} \mathbb{B}[t] | t \in T$ et $\mathbb{B}[V] \stackrel{\mathbb{B}}{=} \mathbb{B}[V_c] \cup \mathbb{B}[V_v] \stackrel{\mathbb{B}}{=} V_c \cup V_v$ (déclaration des constantes et variables)
- $\mathbb{B}[type] \stackrel{\mathbb{B}}{=} \bigwedge \mathbb{B}[x_i] \in \mathbb{B}[t_i] | (x_i, t_i) \in type$ (typage des constantes et variables)

$$\stackrel{\mathbb{B}}{=} \begin{cases} \mathbb{B}[type_c] & \forall (c,t) \in type_c \Rightarrow (c,t) \in type \wedge c \in V_c \\ \mathbb{B}[type_v] & \forall (v,t) \in type_v \Rightarrow (v,t) \in type \wedge v \in V_v \end{cases}$$

- $\mathbb{B}[Inv] \stackrel{\mathbb{B}}{=} \forall pred \in Inv \bullet \bigwedge \mathbb{B}[pred]$ (prédicats de l'invariant)
- $\mathbb{B}[Init] \stackrel{\mathbb{B}}{=} \{(\mathbb{B}[v], \mathbb{B}[expr]) | (v, expr) \in Init\}$ (état initial)

L'ensemble $\mathcal{D}^P$ des services offerts d'un composant C est traduit par un ensemble d'opérations B. $\mathbb{B}[\mathcal{D}^P] \stackrel{\mathbb{B}}{=} \{\mathbb{B}[srv] | srv \in \mathcal{D}^P\}$. Un service est décrit formellement par un triplet $\langle \mathcal{IS}, l\mathcal{W}, \mathcal{B}eh\rangle$. Dans le contexte de la vérification de pré/post-conditions des services par rapport à l'invariant du composant, seule l'interface $\mathcal{IS} = \langle \sigma, Pre, Post\rangle$ est concernée par la traduction [2]. Le schéma de traduction des services est le suivant :

$$\mathbb{B}[srv] \stackrel{\mathbb{B}}{=} \mathbb{B}[\langle \sigma, Pre, Post\rangle] \stackrel{\mathbb{B}}{=} \begin{cases} \mathbb{B}[\sigma] \stackrel{\mathbb{B}}{=} \begin{cases} \mathbb{B}[name] \\ \mathbb{B}[param] \\ \mathbb{B}[ptype] \\ \mathbb{B}[Tres] \end{cases} & \stackrel{\mathbb{B}}{=} \{\ result \leftarrow name(\mathbb{B}[param]) \\ \mathbb{B}[Pre] & \stackrel{\mathbb{B}}{=} \begin{cases} \text{PRE} \\ \quad \mathbb{B}[ptype] \wedge \\ \quad \mathbb{B}[Pre] \end{cases} \\ \mathbb{B}[Post] & \stackrel{\mathbb{B}}{=} \begin{cases} \text{ANY} \\ \quad \mathbb{B}[\alpha_l\_V_{Post}] \cup \{l\_result\} \\ \text{WHERE} \\ \quad \mathbb{B}[\alpha_l\_Post] \wedge \\ \quad \alpha_l\_type(V_{Post}) \wedge \\ \quad l\_result \in \mathbb{B}[Tres] \\ \text{THEN} \\ \quad \mathbb{B}[V_{Post}] := \mathbb{B}[\alpha_l\_V_{Post}] \,||\, \\ \quad result := l\_result \,||\, \\ \quad name\_res := l\_result \end{cases} \end{cases}$$

où $result$ représente la valeur de retour du service, $name$ le nom du service et $\mathbb{B}[param] = param$ définissent la signature de l'opération.

2. L'espace d'état $l\mathcal{W}$ est exploité ultérieurement pour vérifier les liens d'assemblages. Le comportement $\mathcal{B}eh$ est utilisé pour la vérification fonctionnelle du service lui même. Ces aspects ne sont pas traités dans cet article.

La pré-condition de l'opération B assure le typage des paramètres en plus de la pré-condition du service :
$\mathbb{B}[ptype] \stackrel{\mathbb{B}}{=} \bigwedge \mathbb{B}[p_i] \in \mathbb{B}[t_i] | (p_i, t_i) \in ptype \wedge \mathbb{B}[pre] \stackrel{\mathbb{B}}{=} \forall pred \in Pre \bullet \bigwedge \mathbb{B}[pred]$.

Une post-condition Kmelia établit une relation entre les variables avant et après l'exécution. Elle est traduite par une substitution non déterministe sur des variables locales (clause ANY) de même type que les variables modifiées et qui satisfont la post-condition (clause WHERE). Ces variables locales sont affectées aux variables d'état de la machine (clause THEN). Soit α_l_θ une α-conversion qui remplace les occurrences d'une variable v de θ par l_v.

- $\mathbb{B}[\alpha_l\_V_{Post}]$ est un ensemble de variables locales qui représentent les valeurs des variables de la machine après l'exécution de l'opération. Seules les variables modifiées sont concernées.
- $\{l\_result\}$ est une variable locale modélisant la valeur de retour.
- $\mathbb{B}[\alpha_l\_Post] \stackrel{\mathbb{B}}{=} \forall pred \in Post \bullet \bigwedge \mathbb{B}[\alpha_l\_pred]$.
- $\mathbb{B}[\alpha_l\_type] \stackrel{\mathbb{B}}{=} \forall (v, t) \in type \bullet \bigwedge \mathbb{B}[l\_v] \in \mathbb{B}[t]$.
- $name\_res$ représente la variable modélisant la valeur de retour du service $name$ dans le composant. Nous aurions pu utiliser directement le prédicat *before-after* de Event-B, mais dans la version actuelle de l'extraction on génère du B classique.

Assemblages et compositions

Dans cette section nous présentons une solution pour la traduction des assemblages de composants Kmelia en B. Soit (C_p, s_p, C_r, s_r) un lien qui relie un service requis s_r de C_r à un service offert s_p de C_p dans un assemblage donné.

Listing 5 – *Cas d'un lien (s_r, s_p)*

```
MACHINE sr
VARIABLE
      V_sr,  rs
INVARIANT
      B[type(V_sr)]  ∧ B[Inv_sr]  ∧ rs  ∈ B[Tres]
OPERATIONS
    rr  ⟵ sr (P) =
      Pre
         B[type(P)]  ∧  B[Pre_sr]
      ANY
         α_l_V_sr, l_r
      WHERE
         B[α_l_Post_sr] ∧ B[α_l_type(V_sr)] ∧
         l_r  ∈ B[Tres]
      THEN
         V_sr := α_l_V_sr || rs := l_r
END
```

```
REFINEMENT sr_sp_ref
REFINES sr
VARIABLE
      V_sr,  rp ,  V_obs
INVARIANT
      B[type(V_obs)] ∧ B[Inv_obs] ∧
      B[Map(V_sr, Exp(V_obs))] ∧ rp ∈ B[Tres]
OPERATIONS
      rr  ⟵ sr (P) =
         Pre
            B[type(P)]  ∧ B[Pre_obs]
         ANY
            α_l_V_obs, α_l_V_sr, l_r
         WHERE
            B[α_l_Post_sr] ∧ B[Map (α_l_V_sr, Exp(α_l_V_obs))]
            B[α_l_type(V_sr)] ∧ l_r ∈ B[Tres]
         THEN
            V_sr,  V_obs, rp  := α_l_V_sr, α_l_V_obs, l_r
END
```

Le contrat qui établit un assemblage correct en Kmelia est que la pré-condition de s_r est plus forte que celle de s_p et la post-condition de s_r est elle plus faible que celle de s_p.

Pour transposer la garantie de la correction du contrat d'assemblage en B, le principe est de considérer le service offert s_p comme un raffinement du service requis s_r, ce qui conduit à la génération en B d'une obligation de preuve de raffinement similaire à celle posée en Kmelia.

La mise en œuvre nécessite la génération d'une machine sr à partir du service requis s_r et un raffinement sr_sp_ref qui raffine la machine sr en décrivant le service offert s_p. Le listing 5 montre la structure des machines B générées.

Pour chaque service requis s_r dans un composant C, une machine sr devait être créée auparavant pour vérifier la cohérence du contexte virtuel du service s_r. C'est cette même machine que nous raffinons par sr_sp_ref .
- L'espace d'état de la machine sr est obtenu par la traduction du contexte virtuel du service s_r, et l'opération rr est la traduction du service s_r.
- Dans le raffinement sr_sp_ref les variables observables du service s_p sont réunies et l'invariant est complété par le prédicat de collage Map qui exprime le *mapping* entre les variables virtuelles du service requis s_r et les variables observables du service offert s_p ;
- $\mathbb{B}[Map(\alpha_l\_V_{sr}, Exp(\alpha_l\_V_{obs}))] \stackrel{\mathbb{B}}{=} \forall(v, E(V_{obs})) \in Map \bullet \bigwedge l\_v = \mathbb{B}[E(\alpha_l\_V_{obs})]$, où $E(V_{obs})$ est une expression Kmelia sur les variables observables de s_p.

6 Expérimentations

Cette section commente quelques résultats obtenus en appliquant les règles de transformation de la section 5 sur l'exemple de la figure 1. Les expérimentations ont été réalisées avec la plateforme COSTO et l'Atelier B [3]. La plateforme COSTO est un plugin Eclipse et l'outil Kml2B présenté ici fait l'objet d'un plugin dépendant de COSTO. Dans un premier temps les spécifications Kmelia sont analysées avec COSTO (vérifications de syntaxe, de type et de cohérence simple). Puis les machines B sont extraites par appels de la fonction Kml2B que nous avons développée en fonction des propriétés à vérifier (cf tableau 1).

(PI)	machine StockManager	**(PO)**	machine StockManager_obs
(PV)	machine addItem_mch	**(CA)**	raffinement v_addItem_sm_newReference

L'analyse de cet exemple avec l'Atelier B a révélé un certains nombre d'erreurs, par exemple la précondition réduite à $True$ dans le service requis addItem a violé le contrat d'assemblage $(Pre(addItem) \Rightarrow Pre(newReference))$ car $(True \Rightarrow not(size(catalog) < MaxRef))$ n'est pas toujours vrai. Dans ce cas, deux solutions sont possibles pour corriger cette erreur, soit affaiblir la précondition du service newReference, soit renforcer celle du service addItem. Dans un premier temps, nous avons choisi la première solution, la précondition du service newReference a été affaiblie à $True$. Cela nous a permis de prouver la correction du lien d'assemblage mais pas l'assemblage, car le composant StockManager n'est désormais plus cohérent. Cela est dû au fait que son invariant $size(catalog) <= MaxRef$ n'est plus préservé par la post-condition. La deuxième solution a consisté à renforcer la précondition du service addItem par le prédicat $not\ (catalogFull)$. Ainsi nous avons prouvé l'assemblage tout en préservant la cohérence du composant StockManager.

Après la correction de ces erreurs, les machines B extraites à partir de la nouvelle spécification Kmelia ont généré 28 obligations de preuve qui ont été prouvées par le prouveur de l'atelier B en mode *Automatic (force 1)*.

Traçabilité

L'interprétation des erreurs obtenues sur la spécification Kmelia de départ nous a été largement facilitée par la formalisation des règles de transformations. Considérons le message d'erreur $msg = (Inv_C, reference, Pred)$ que le prouveur de l'atelier B génère après échec de preuve d'une obligation de preuve, où Inv_C est un prédicat de l'invariant de la machine

3. http://www.aterlierb.eu

C en cours de vérification, $reference$ est une référence soit à une initialisation soit à une opération s et $Pred$ est un prédicat d'une substitution dans $reference$ qui ne respecte pas l'invariant. Nous améliorons la traçabilité des erreurs de spécification par une génération de messages d'erreurs $msg = (\mathbb{B}^{-1}[Inv_C], reference, \mathbb{B}^{-1}[Pred])$ de la forme suivante :

> *"In a Kmelia component C, check that the invariant $(\mathbb{B}^{-1}[Inv_C])$ is established by the service s"*
> => $\mathbb{B}^{-1}[Pred]$

Cette gestion des erreurs permet d'assister le spécifieur Kmelia et de rendre l'utilisation de B plus transparente.

7 Travaux connexes

Parmi les travaux consacrés à la formalisation ou la vérification de propriétés sur des modèles à composants, un bon nombre s'attache à la formalisation ou au raisonnement sur les modèles UML, en utilisant des techniques et outils formels comme B ou Alloy pour exprimer des contraintes (souvent OCL) incorporées dans les modèles. Dans la plupart de ces travaux, une partie du modèle d'entrée UML (incluant des propriétés en OCL) est traduite en B comme dans Marcano-Kamenoff et Lévy (2001); Ledang et Souquières (2002) ou en Alloy dans Anastasakis et al. (2007). Nous avons d'une part une démarche plus globale prenant tout le modèle d'entrée et d'autre part notre travail est positionné au niveau de la spécification des composants et de leurs assemblages. Dans Snook et Butler (2006) les auteurs utilisent un dialecte B comme langage d'expression de données et de contraintes pour UML ; ils dérivent ainsi aisément une sémantique des entités UML par traduction en B et proposent un profil UML-B pour permettre le raffinement des modèles UML. En fin de compte, l'approche adoptée par les auteurs est vue comme une une variante formelle de UML permettant le raisonnement formel à la façon de B. Notre démarche n'est pas comparable avec celle proposée dans ce travail cependant les objectifs se rejoignent sur le plan de la vérification de propriéts de modèles (à composants).

Un travail similaire au notre est présenté dans Bouquet et al. (2007) où les auteurs ont expérimenté la preuve de modèles JML (code Java annoté par des assertions en logique du premier ordre) en utilisant la méthode B. Nous partageons les mêmes objectifs de preuve de modèles spécifiques mêmes si les modèles de départ sont très différents. Au niveau des composants élémentaires les traductions en B que nous avons faites sont semblables à celles faites des classes/objets JML vers B. Mais en ce qui concerne les assemblages, nous avons une approche basée sur les raffinements qui n'a pas d'équivalent dans l'approche avec JML même pour le traitement de l'héritage qui aurait pu être exprimé par un raffinement des méthodes/opérations où on renforcerait les préconditions des opérations. Notre travail est comparable à ceux de Chouali et al. (2006); Colin et al. (2009) où les auteurs proposent une méthode pour développer et vérifier l'assemblage des composants via leurs interfaces à l'aide de la méthode B ; néanmoins leur approche suppose la cohérence préalable de leurs interfaces et composants. De plus le détail des composants est écrit en B, notre approche est donc plus indépendante. Elle est aussi plus complète dans le sens où on vérifie d'abord la cohérence des composants primitifs et puis la correction des assemblages, y compris pour les communications, même si ce n'est pas l'objet de cet article.

En résumé nos travaux se situent dans une lignée de travaux sur la réutilisation des plateformes de vérification formelle, pour établir la correction de modèles à composants, mais avec

des spécificités telles que l'approche globale de correction par construction qui est mise en œuvre très tôt dans la construction des composants.

8 Bilan et perspectives

Dans cet article nous avons montré comment vérifier la cohérence de composants et assemblages Kmelia en se servant d'une plateforme de preuve dédiée à la méthode B. La transposition de la vérification a été possible parce que nous avons adopté dans Kmelia une approche de vérification basée sur des contrats Pré/Post (aussi bien pour les services des composants que pour leurs assemblages) qui s'avèrent être compatibles avec les obligations de preuve de cohérence des spécifications B. Nous extrayons alors systématiquement, sur la base de règles de traduction, des spécifications B à partir des composants ou assemblages Kmelia. Nous avons développé comme extension de la plateforme COSTO, un module qui effectue cette traduction. Les machines B résultant des traductions sont intégrées dans l'Atelier B et soumises à la preuve. Les expérimentations ont montré que les erreurs dans les spécifications Kmelia se traduisent par des obligations de preuve non déchargées. Elles peuvent alors être corrigées.

Un travail entrepris pour le court terme est l'aide à la rétroaction sur les spécifications Kmelia ; il s'agit d'exploiter les résultats des analyses en B pour référencer les parties erronées dans la spécification Kmelia de départ. La proximité des structures de Kmelia avec celle de B et les règles de traduction définies devraient aider à obtenir rapidement des résultats dans ce sens. Pour compléter les propriétés prouvées sur les spécifications Kmelia dans le but d'obtenir des composants et assemblages corrects, nous travaillons sur la preuve de correction fonctionnelle des services. Ici, étant donné un service spécifié par ses pré/post-conditions, nous voulons prouver que le comportement (sous forme d'automate) donné au service est en adéquation avec les pré/post-conditions. Parmi les pistes que nous explorons, il y a le calcul de la précondition en partant de la post-condition et en remontant l'automate qui décrit le système ; nous explorons aussi une conversion en Event-B de l'automate de comportement puis une preuve de cohérence par rapport à l'invariant quand le déroulement du service arrive à l'état final.

Références

Abrial, J.-R. (1996). *The B-Book Assigning Programs to Meanings*. Cambridge University Press.

Abrial, J.-R., M. Butler, S. Hallerstede, et L. Voisin (2006). An Open Extensible Tool Environment for Event-B. In *ICFEM 2006*, Volume 4260 of *LNCS*. Springer.

Abrial, J.-R. et S. Hallerstede (2007). Refinement, Decomposition, and Instantiation of Discrete Models : Application to Event-B. *Fundamenta Informaticae 77*(1-2), 1–28.

Allen, R. et D. Garlan (1997). A Formal Basis for Architectural Connection. *ACM TOSEM 6*(3), 213–249.

Anastasakis, K., B. Bordbar, G. Georg, et I. Ray (2007). UML2Alloy : A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, Volume 4735 of *LNCS*. Springer.

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

André, P., C. Attiogbé, et M. Mohamed (2009b). Correction d'assemblages de composants impliquant des interfaces paramétrées. In *3e Conférence Francophone sur les Architectures Logicielles*, Volume RNTI-L-4 of *Revue des Nouvelles Technologies de l'Information*, pp. 33–44. Cépaduès-Éditions.

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.

Bouquet, F., F. Dadeau, et J. Groslambert (2007). Checking JML Specifications with B Machines. In *B'2007*, Volume 4355 of *LNCS*, Besancon, France, pp. 285–288. Springer.

Chouali, S., M. Heisel, et J. Souquières (2006). Proving component interoperability with B refinement. *Electronic Notes in Theoretical Computer Science 160*, 157–172.

Clements, P. C. (1996). A Survey of Architecture Description Languages. In *IWSSD '96 Proceedings*, Washington, DC, USA, pp. 16. IEEE Computer Society.

Colin, S., A. Lanoix, et J. Souquières (2009). Trustworthy interface compliancy : Data model adaptation using b refinement. *Electron. Notes Theor. Comput. Sci. 203*(7), 23–35.

Ledang, H. et J. Souquières (2002). Integration of UML and B Specification Techniques : Systematic Transformation from OCL Expressions into B. In *9th Asia-Pacific Software Engineering Conference (APSEC 2002)*. IEEE Computer Society.

Marcano-Kamenoff, R. et N. Lévy (2001). Transformation d'annotations OCL en expressions B. In *AFADL'2001*, pp. 39–49.

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

Meyer, B. (2003). The grand challenge of Trusted Components. In *ICSE '03 Proceedings*, Washington, DC, USA, pp. 660–667. IEEE Computer Society.

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.

Snook, C. et M. Butler (2006). UML-B : Formal Modeling and Design aided by UML. *ACM Trans. Softw. Eng. Methodol. 15*(1), 92–122.

Summary

To build Component Based Software Systems with quality assessment for trusted components and assemblies, we need methods and tool supports to prove correctness. Kmelia is a multi-service component model where the components are defined in an abstract and formal manner. Hence specification properties can be expressed and proved. In this article we study the automated checking of the consistency of Kmelia components and assemblies using the B method. We extract B machines from Kmelia specifications in such a way that we can check the consistency and also the correctness of assembly at the Kmelia level, using B provers. An illustrative example based on a stock management system is used to support the study.