Spécification de composants pour assurer leur interopérabilité

Maritta Heisel
Fakultät für Informatik
Universität Magdeburg
D-39016 Magdeburg, Germany
heisel@cs.uni-magdeburg.de

Jeanine Souquières
LORIA—Université Nancy2
B.P. 239 Bâtiment LORIA
F-54506 Vandœuvre-les-Nancy, France
Jeanine.Souquieres@loria.fr

Résumé : Cet article présente une approche pour la spécification de composants logiciels en vue d'assurer leur interopérabilité. Notre contribution concerne les trois points suivants. Elle propose une manière de spécifier les composants indépendamment d'un langage d'expression. Elle met en avant l'importance des données dans la définition de composants en rajoutant aux contrats d'utilisation entre composants un modèle de données. Elle introduit une notion de compatibilité entre interfaces pour décider de l'interopérabilité entre composants.

Mots clés : Composants, spécification, interopérabilité, modèle de données, interface d'exportation, interface d'importation.

1 Introduction

L'ingénierie du logiciel basée sur les composants est actuellement d'un grand intérêt aussi bien dans le domaine de la recherche que dans la pratique [Szy99]. Elle est issue des approches orientées objet [HC01]. L'objectif est de développer des systèmes par assemblage de composants comme cela est réalisé dans d'autres disciplines qui proposent des standards pour décrire les composants ainsi que des catalogues de composants. Les composants sont des logiciels construits de manière indépendante. Ils ont les caractéristiques suivantes:

- les services offerts par un composant et les services que ce composant utilise sont accessibles via ses interfaces,
- plusieurs modèles de composants sont proposés tels les JavaBeans [Sun97], Entreprise Java Beans [Sun01], Microsoft COM [Mic95] et CORBA [Obj98]. Un modèle de composants a pour objectif l'interopérabilité de composants implantés suivant des standards proposés par le modèle. Ainsi, construire un système à l'aide de composants signifie que l'on sélectionne des composants conformes à un modèle particulier de composants et qu'on les compose pour décrire le comportement attendu du système,
- les composants sont implantés mais l'accès au code source n'est pas toujours possible. C'est pourquoi la description de leurs interfaces joue un rôle important dans un développement axé composants [HSS02].

La spécification d'un composant doit contenir toutes les informations nécessaires pour assurer son utilisation dans un contexte donné. Un composant peut être utilisé directement pour décrire le système attendu ou bien il peut être composé avec un ou plusieurs autres composants. Dans ce cas, il faut connaître l'ensemble des informations nécessaires à leur coopération. Quand peut-on accepter ou refuser un composant en vue de l'intégrer dans une nouvelle architecture? Cette

question est reliée au problème connu d'interopérabilité: comment s'assurer de l'usage correct d'un composant par le système utilisateur et comment s'assurer que les ressources requises par ce composant sont bien disponibles?

Pour cela, un composant offre des interfaces d'importation et des interfaces d'exportation, un contrat d'utilisation précis entre composants et un protocole d'utilisation qui gère la communication entre composants. Notre proposition consiste à rajouter à ce contrat d'utilisation un modèle de données permettant d'exprimer l'interopérabilité entre composants via la compatibilité de leurs interfaces. Elle peut être combinée avec l'approche modèle de composants mais elle est indépendante. Dans cet article, nous nous intéressons uniquement aux aspects fonctionnels d'un composant, en sachant que des aspects non-fonctionnels comme la sécurité de fonctionnement et la sûreté jouent aussi un rôle crucial dans l'utilisation de composants. Nous proposons:

- une manière de spécifier les composants indépendamment d'un langage d'expression,
- la mise en avant de l'importance des données dans la spécification d'un composant,
- la notion de compatibilité entre les interfaces des différents composants car elles ne sont pas forcément identiques.

Les protocoles d'utilisation des composants définissent généralement les traces permises, c'est à dire les différents enchaînements permis entre les différentes opérations. Reussner [Reu01] propose une approche sophistiquée en exprimant les protocoles à l'aide d'automates paramétrés. Par contre, ici, nous utilisons une manière plus simple, à savoir un ordre de dépendance entre les opérations qui correspond au "life cycle model" dans la méthode Fusion [CAB+94].

Cheesman et Daniels [CD01] proposent un processus pour spécifier un système basé sur des composants. Le processus commence avec une description informelle et produit une architecture mettant en évidence les composants à développer ou à réutiliser, leurs interfaces et leurs dépendances. L'approche suit les principes de la construction par contrats [Mey97]. La méthode de Cheesman et Daniels se termine avec la spécification des composants; elle ne considère pas le passage de la spécification des composants vers un modèle de composants concrets ni l'implantation des composants spécifiés et de leurs interfaces. La spécification d'un composant est constituée de l'identification du composant avec le développement d'un modèle de données et l'identification des types essentiels. Un composant est développé pour le système ainsi que pour chaque type essentiel. Les opérations de chaque interface sont identifiées avec leurs paramètres. Des diagrammes de collaboration sont ensuite développés montrant comment chaque opération de l'interface du système à définir doit interagir avec les autres composants de l'architecture pour remplir son objectif. Ces diagrammes de collaboration mettent en évidence les opérations des interfaces des autres composants qui sont nécessaires. Pour terminer, les opérations sont spécifiées en termes de pré et de postconditions. Les interfaces d'importation ne sont pas spécifiées dans cette approche, et les protocoles d'utilisation ne sont pas considérés. L'utilisation de cette méthode conduit à la génération d'un certain nombre de documents décrits en UML [RJB97] et en OCL [WK99].

Notre approche couvre celle de Cheesman et Daniels; elle rajoute le traitement systématique des interfaces d'importation et la notion de compatibilité entre différents composants.

Overhagen [Ove03] propose un cadre de développement de composants avec trois différentes vues, une vue statique décrivant le modèle de données, une vue opérationnelle et une vue dynamique pour exprimer les dépendances entre les différentes opérations. Outre les aspects fonctionnels, il s'intéresse à d'autres aspects, par exemple la qualité de service et l'information de nature commerciale. Overhagen ne considère pas la compatibilité entre différents composants.

L'article est structuré comme suit. Le chapitre 2 présente la structure des spécifications avec la notion de modèle de données et les interfaces d'importation et d'exportation. Le chapitre 3

présente la définition de la notion de compatibilité entre interfaces. L'exemple utilisé tout au long de ce papier est une variante du système de réservation d'hôtels proposé par Cheesman et Daniels [CD01]. Une conclusion avec des perspectives d´évolution termine ce papier.

2 Structure des spécifications

Notre objectif est de proposer une manière de spécifier des composants permettant de décider si un composant peut utiliser un autre composant sans regarder son code. Dans l'approche basée composants on n'est pas sûr d'avoir accès au code d'un composant. C'est pourquoi les spécifications, en particulier les spécifications des interfaces, jouent un rôle très important.

2.1 Définitions

La spécification doit contenir tous les aspects cruciaux pour décider si un composant peut être utilisé dans un contexte donné. Cela concerne les données utilisées par le composant ainsi que son comportement visible par son environnement. Ce comportement est exprimé à l'aide de services que le composant met à la disposition d'autres composants ou d'autres logiciels. Ces services sont nommés *interfaces d'exportation*. Dans la plupart des cas, cependant, un composant dépend de services rendus par d'autres composants. Il peut fonctionner correctement seulement avec la présence d'autre composants qui rendent des services requis. De telles contraintes sont nommées *interfaces d'importation*. Elles font aussi partie de la spécification d'un composant, même si ces interfaces ne sont pas présentes dans les modèles de composants comme Enterprise Jave Beans [Sun01] ou le modèle de composants de Microsoft [Mic95]. Cheesman et Daniels [CD01] mentionnent ces deux sortes d'interfaces, mais ne spécifient pas les interfaces d'importation dans leur approche.

2.1.1 Spécification d'un composant

Un composant est constitué de plusieurs ingrédients nécessaires à son utilisation. Ces ingrédients sont:

1. la spécification des interfaces d'exportation,
2. la spécification des interfaces d'importation,
3. les dépendances entre les interfaces d'exportation nécessaires à l'utilisation du composant.
   Un composant peut avoir plusieurs interfaces d'exportation qui rendent des services différents à son environnement. Ces services ne sont cependant pas toujours indépendants les uns des autres. Les interfaces d'exportation peuvent correspondre aux différents cas d'utilisation, comme par exemple ouvrir un compte dans une banque ou effectuer une transaction sur un compte bancaire existant. Bien sûr, ces cas d'utilisation (et, par conséquent, les interfaces) ne sont pas indépendants, parce qu'il faut ouvrir le compte avant d'effectuer des transactions. Les dépendances entre les interfaces d'exportation décrivent l'ordre dans lequel les différentes interfaces doivent être utilisées. En ce sens, elles constituent un mode d'emploi pour le composant.
   L'expression $EI_1 \prec EI_2$ exprime qu'avant d'utiliser des opérations de l'interface d'exportation EI_1, il faut utiliser des opérations de l'interface d'exportation EI_2.
   Nous avons choisi d'exprimer les dépendances au niveau des interfaces et non au niveau des opérations, parce que les interfaces sont les plus petites unités d'utilisation (ou de réutilisation) d'un composant,
4. les dépendances entre les interfaces d'exportation et les interfaces d'importation.
   Ces dépendances (notées $EI \prec II$) permettent de savoir quelle interface d'importation est nécessaire pour rendre les services d'une interface d'exportation. Un composant peut être

utilisé même si toutes ses interfaces d'importation ne sont pas disponibles, mais avec un service réduit.

La spécification des interfaces est décrite plus en détails dans le paragraphe suivant.

2.1.2 Spécification d'une interface

La spécification d'une interface est constituée d'un modèle de données appelé IDM (Interface Data Model), d'un ensemble d'opérations et d'un protocole d'utilisation des différentes opérations.

Spécification d'un modèle de données La spécification du modèle de données est constituée des types utilisés, de la spécification de la structure d'état des données présentes et des invariants. On peut la représenter sous la forme d'un diagramme de classes avec des contraintes exprimées en OCL, ou bien à l'aide de langages de spécification comme Object-Z [Smi99].

Spécification d'une opération Pour décrire un composant, la spécification d'une opération est constituée de son profil, d'une précondition et d'une postcondition, ces deux derniers ingrédients faisant référence à l'IDM correspondant.

Spécification du protocole d'utilisation Ce protocole doit spécifier l'ordre permis entre les appels des opérations. Pour l'exprimer, il y a plusieurs possibilités. Par exemple, dans l'étude de cas présentée dans ce papier, il suffit de donner des dépendances simples. L'expression $op_1(x) \prec op_2(y)$ exprime qu'avant d'appeler l'opération op_1 avec son paramètre x, il faut appeler l'opération op_2 avec son paramètre y. Chaque suite d'opérations doit obéir à ces contraintes. Cette manière simple de décrire les protocoles d'utilisation suffit pour un nombre important de systèmes.

Dans le cas où l'expression de contraintes plus complexes est nécessaire, on peut utiliser des algèbres de processus (par exemple, CSP [Hoa85]), ou bien des automates [Reu01].

2.2 Étude de cas

Soit une variante du système de réservation d'hôtels proposé par Cheesman et Daniels [CD01]. Notre objectif est de décrire un tel système en utilisant des composants existants comme par exemple le composant *HotelMgr* qui permet de gérer des chambres d'hôtel dans lequel différents types de chambres sont proposées.

Le système global de réservations dispose de plusieurs interfaces d'importation et d'exportation. La figure 1 présente deux interfaces d'exportation à savoir *IMakeReservation* et *ITakeUp−Reservation* et deux interfaces d'importation concrètes à savoir *IHotelHandling* et *ICustomer−Handling* du composant *ReservationSystem*. Il existe plusieurs dépendances entre ces interfaces:

- *ITakeUpReservation* $\prec$ *IMakeReservation*, parce qu'il est nécessaire d'effectuer une réservation avant de la "prendre".
- *IMakeReservation* $\prec$ *IHotelHandling* et *ITakeUpReservation* $\prec$ *IHotelHandling*, parce que sans la gestion des hôtels, il est impossible d'effectuer des réservations.
- *IMakeReservation* $\prec$ *ICustomerHandling* et *ITakeUpReservation* $\prec$ *ICustomerHandling*

Dans la suite, on décrira uniquement les interfaces *IMakeReservation* et *IHotelHandling* du composant.

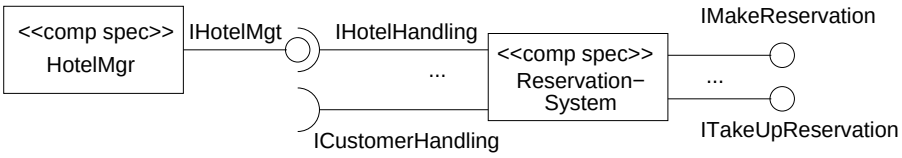

FIG. 1 – *Architecture de composants du système de réservations d'hôtels*

2.2.1 Spécification de l'interface d'exportation *IMakeReservation*

La figure 2 présente le modèle de données du composant *ReservationSystem* pour son interface d'exportation *IMakeReservation* ainsi que les types essentiels pour décrire les opérations de cette interface.

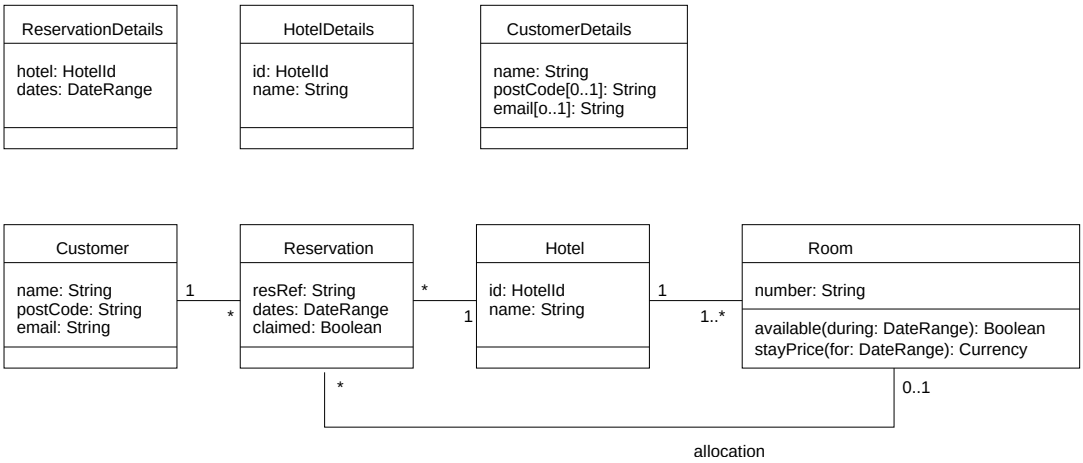

FIG. 2 – *Modèle de données pour l'interface d'exportation IMakeReservation du composant ReservationSystem*

Deux invariants ont été introduits: le premier précise que le nombre maximum de réservations pour une date donnée est inférieur ou égal au nombre de chambres disponibles dans l'hôtel et le deuxième indique qu'une réservation n'est effectuée que si on lui a associé une chambre.

```
Date->forAll(d | reservation->select(r | not r.claimed and
                                r.dates.includes(d))->size) <= room->size
```

```
Reservation
claimed = allocation->notEmpty
```

L'interface *IMakeReservation* propose trois opérations, *getHotelDetails*, *getRoomInfo* et *make−Reservation* dont les définitions OCL sont données ci-dessous. L'opération *getHotelDetails* renvoie la liste des hôtels disponibles qui correspondent au paramètre donné *match* avec pour chacun d'eux un certain nombre de détails comme son identificateur et son nom.

```
getHotelDetails(in match: String): HotelDetails[]
pre:  true
post: -- give details for all hotels whose name matches the input string
      result->forAll(hd | hotel->exists(h | h.name = hd.name and
                                            h.id = hd.id and
                                            matches(match, h.name)))
```

L'opération *getRoomInfo* renvoie des informations sur la disponibilité et le prix des chambres de l'hôtel correspondant à la réservation donnée.

```
getRoomInfo(in res: ReservationDetails, out availability: Boolean,
            out price: Currency)
pre:  -- the hotel must exist
      hotel.id->includes(res.hotel)
post: -- rms is the set of all rooms of the chosen hotel
      -- that are available at the given date range
      let rms = hotel.room->select(x | x.hotel.id = res.hotel and
                                       x.available(res.dates) = true)
      in if   rms->notEmpty
         then availability = true and
              price = rms->asSequence->first.stayPrice(res.dates))
         else
         availability = false
```

L'opération *makeReservation* effectue une réservation pour un hôtel donné seulement si le client est répertorié (voir la description d'un client dans la classe *Customer*).

```
makeReservation(in res: ReservationDetails,
    in cus: CustomerDetails, out resRef: String): Integer
pre:
  -- the hotel specified is valid
     hotel.id -> includes(res.hotel)
post:
  result=0 implies
       -- a reservation was created
       -- select hotel
       let h = hotel -> select(x | x.id=res.hotel)
          -> asSequence -> first in
          -- there exists one more reservation than before
         (h.reservation - h.reservation@pre) -> size=1 and
         -- select new reservation
         let r = (h.reservation - h.reservation@pre)
                  -> asSequence -> first
         -- properties of new reservation
         in r.resRef = resRef and
            r.dates = res.dateRange and
            not r.claimed and
            r.customer.name = cus.name and
            cus.postCode -> notEmpty implies
                      cus.postCode = r.customer.postCode and
            cus.email -> notEmpty implies
                      cus.email = r.customer.email
       -- result=1 implies customer not found and unable
                  to create
       -- result=2 implies more than one matching customer
```

Les dépendances entre ces différentes opérations doivent être précisées: avant d'appeler l'opération *makeReservation*, il faut appeler l'opération *getRoomInfo* au moins une fois, et avant d'appeler l'opération *getRoomInfo*, il faut appeler l'opération *getHotelDetails* au moins une fois:

$makeReservation \prec getRoomInfo$

$getRoomInfo \prec getHotelDetails$

2.2.2 Spécification de l'interface d'importation *IHotelHandling*

La figure 3 présente le modèle de données du composant *ReservationSystem* pour son interface d'importation *IHotelHandling* ainsi que les types essentiels pour décrire les opérations de cette interface. Par rapport à la description proposée dans le paragraphe 2.2.1, on n'a plus besoin du type *CustomerDetails*, mais on utilise des identificateurs pour les clients (attribut *Id* dans la classe *Customer* du diagramme de classes).

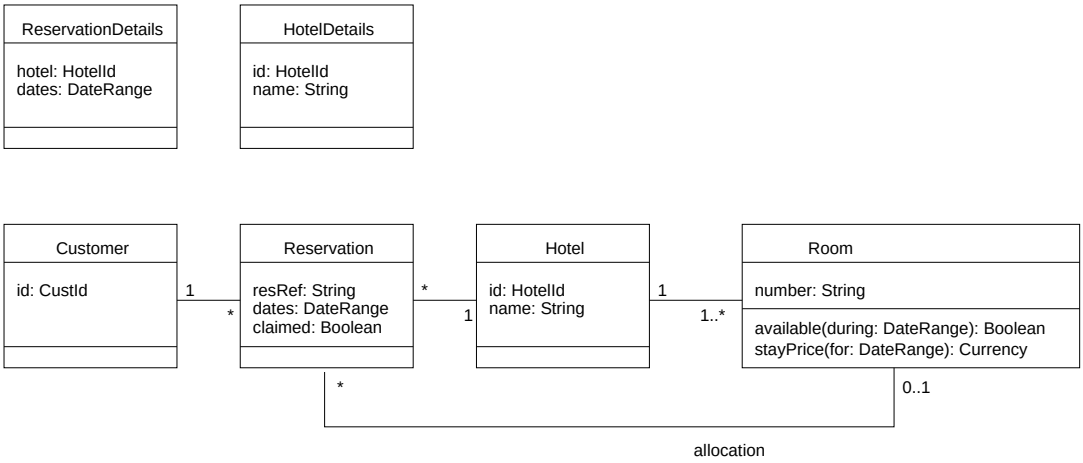

FIG. 3 – *Modèle de données pour l'interface d'importation IHotelHandling du composant ReservationSystem*

Les invariants sont les mêmes que pour l'interface *IMakeReservation*.

L'interface *IHotelHandling* contient trois opérations, *getHotelDetails*, *getRoomDetails* et *make–Reservation* qui correspondent aux opérations de l'interface *IMakeReservation*. Les diagrammes de collaboration des figures 4 et 5 montrent comment les opérations de l'interface *IHotelHandling* sont utilisées pour réaliser les opérations de l'interface *IMakeReservation*.

Il en résulte que les spécifications des opérations *getHotelDetails* et *getRoomInfo* sont les mêmes que dans l'interface *IMakeReservation*. La spécification de l'opération *makeReservation* est un peu différente, parce que les *CustomerDetails* sont remplacés par un *CustId*. La précondition reste la même, mais dans la postcondition, les détails sur le client sont supprimés. De plus, cette opération est appelée seulement si le client existe ou s'il a été créé avec succès lors de l'appel de l'opération.

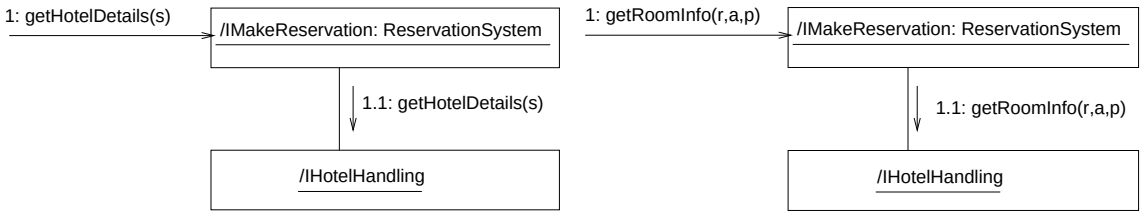

FIG. 4 – *Diagrammes de collaboration des opérations getHotelDetails et getRoomDetails de l'interface IMakeReservation*

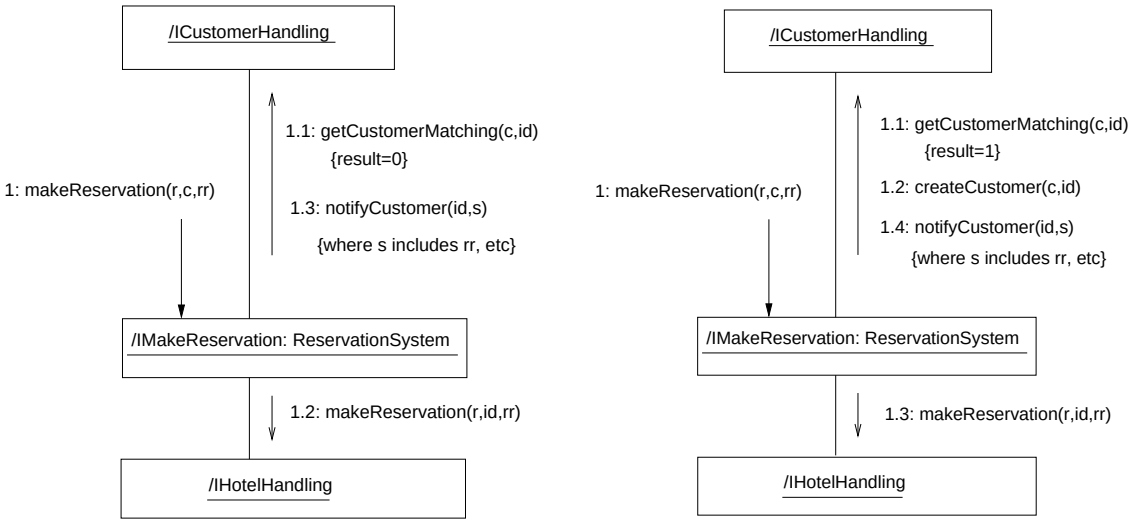

FIG. 5 – *Diagrammes de collaboration de l'opération makeReservation de l'interface IMakeReservation*

```
makeReservation(in res: ReservationDetails,
    in cus: CustId, out resRef: String)
pre:
  -- the hotel specified is valid
     hotel.id -> includes(res.hotel) and
post:
       -- a reservation was created
       -- select hotel
       let h = hotel -> select(x | x.id=res.hotel)
          -> asSequence -> first in
          -- there exists one more reservation than before
         (h.reservation - h.reservation@pre) -> size=1 and
         -- select new reservation
         let r = (h.reservation - h.reservation@pre)
                  -> asSequence -> first
         -- properties of new reservation
         in r.resRef = resRef and
            r.dates = res.dateRange and
            not r.claimed and
            r.customer.id = cus
```

Les dépendances entre les opérations de l'interface *IHotelHandling* sont les mêmes que celle des opérations de l'interface *IMakeReservation*.

2.2.3 Spécification de l'interface *IHotelMgt*

Le composant *HotelMgr* disponible a été développé pour des hôtels offrant plusieurs types de chambres. La figure 6 présente son modèle de données pour son interface d'exportation ainsi que les types essentiels dans les opérations de cette interface. Le modèle de données est similaire à celui du composant *ReservationSystem* avec en plus la classe *RoomType*.

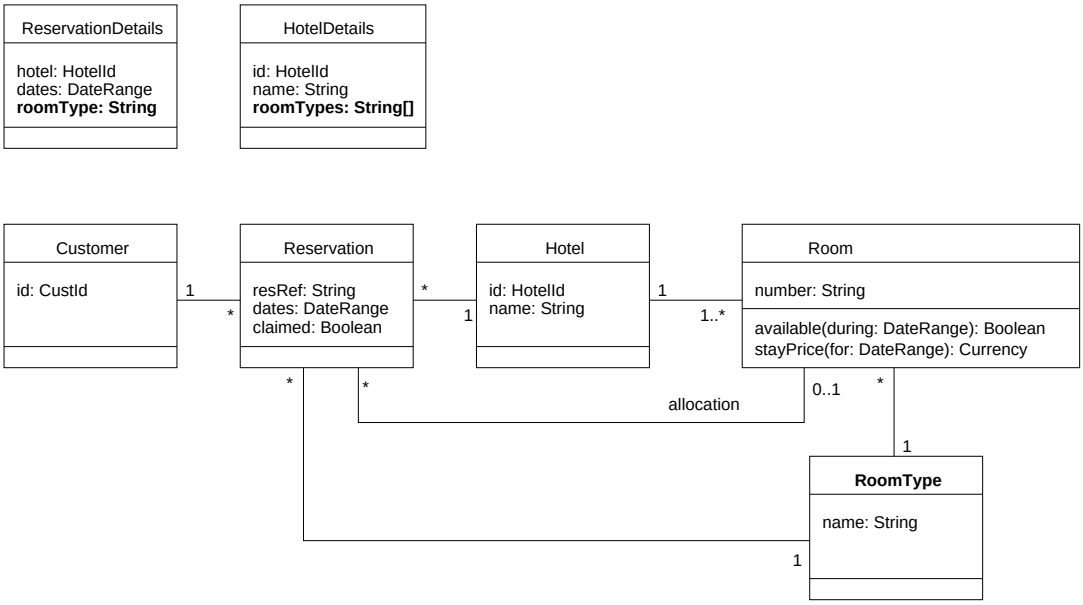

FIG. 6 – *Modèle de données pour l'interface d'exportation du composant HotelMgr*

Les invariants sont les mêmes que pour l'interface *IMakeReservation*. En plus, on exige que pour chaque réservation, le type de chambre soit disponible dans l'hôtel choisi:

```
Reservation
hotel = roomType.room.hotel
```

Les opérations de cette interface prennent en compte les différents types de chambres. Dans la spécification OCL, les différences avec les opérations de l'interface *IHotelHandling* sont notées en gras.

```
getHotelDetails(in match: String): HotelDetails[]
pre:  true
post: -- give details for all hotels whose name matches the input string
      result->forAll(hd | hotel->exists(h | h.name = hd.name and
                                            h.id = hd.id and
                                            h.room.roomType.name = hd.roomTypes
                                            and matches(match, h.name)))

getRoomInfo(in res: ReservationDetails, out availability: Boolean,
            out price: Currency)
pre:  -- the hotel and the room type must exist
      hotel.id->includes(res.hotel) and
      hotel.room.roomType.name->includes(res.roomType)
post: -- rms is the set of all rooms of the chosen hotel
      -- that are available at the given date range
      -- and conform to the specified room type
      let rms = hotel.room->select(x | x.hotel.id = res.hotel and
                                       x.roomType.name = res.roomType and
                                       x.available(res.dates) = true)
      in if   rms->notEmpty
         then availability = true and
              price = rms->asSequence->first.stayPrice(res.dates))
         else
         availability = false
```

```
makeReservation(in res: ReservationDetails,
    in cus: CustId, out resRef: String)
pre:
  -- the hotel and room type specified are valid
     hotel.id -> includes(res.hotel) and
     hotel.room.roomType.name -> includes(res.roomType)
post:
       -- a reservation was created
       -- select hotel
       let h = hotel -> select(x | x.id=res.hotel)
          -> asSequence -> first in
          --there exists one more reservation than before
         (h.reservation - h.reservation@pre) -> size=1 and
         -- select new reservation
         let r = (h.reservation - h.reservation@pre)
                  -> asSequence -> first
         -- properties of new reservation
         in r.resRef = resRef and
            r.dates = res.dateRange and
            r.roomType.name = res.roomType and
            not r.claimed and
            r.customer.id = cus
```

Les dépendances entre les opérations de l'interface *IHotelMgt* sont les mêmes que celle des opérations de l'interface *IMakeReservation*.

3 Notion de compatibilité entre interfaces

Avec la notion de compatibilité entre interfaces, on souhaite garantir qu'une interface d'exportation *EI* d'un composant C' puisse jouer le rôle de l'interface d'importation *II* d'un composant *C*. Cette notion facilite la construction et la modification de systèmes basés sur les composants.

Pour la construction de systèmes à partir de composants, il faut pouvoir combiner plusieurs composants de manière qu'ils puissent inter-opérer. L'interopérabilité s'effectue via des interfaces. On cherche donc à définir une notion de compatibilité basée sur les interfaces et suffisante pour garantir l'interopérabilité entre composants.

Avec une telle notion, on peut aussi déterminer s'il est possible de remplacer un composant par un autre, ce qui facilite la modification de systèmes basés sur les composants.

Par conséquent, la notion de compatibilité est au cœur de l'approche basée composants, parce qu'elle garantit que ses avantages les plus importants (à savoir l'utilisation d'un composant sans en connaître tous ses détails et sa modification) peuvent vraiment être exploités.

La notion de compatibilité entre interfaces doit être définie le plus indépendamment possible de la représentation des interfaces choisie, que l'on utilise des diagrammes de classes tels que proposés dans UML, Object-Z ou encore des spécifications algébriques. Il en est de même pour la représentation des protocoles.

3.1 Définitions

L'interface d'exportation *EI* du composant C' peut jouer le rôle de l'interface d'importation *II* du composant *C*, $II \lhd EI$, si leurs IDMs, leurs opérations et leurs protocoles sont tous compatibles.

3.1.1 Compatibilité entre IDMs

L'idée de base de la compatibilité entre IDMs est que le modèle de données de *II* n'est pas plus restrictif que celui de *EI*. Dans ce cas seulement l'IDM de *EI* peut être utilisée à la place de l'IDM

de *II*. En particulier, chaque type (ou classe) présent dans *II* doit avoir une contrepartie dans *EI*, mais pas forcément l'inverse.

Regardons les différents cas possibles:

- pour les types de base, soit ils ont le même nom, soit on a une règle explicite, e.g. $\mathbb{Z} \lhd \mathbb{N}$ (où $\lhd$ dénote non seulement la relation de compatibilité entre interfaces, mais aussi la compatibilité de leurs ingrédients),
- pour les types structurés, dans le cas des diagrammes de classes: pour chaque classe $classe_1$ de l'IDM de *II*, il existe une classe $classe_2$ de l'IDM de *EI* telle que $classe_1 \lhd classe_2$ avec
  - $classe_1 \lhd classe_2 \Leftrightarrow$
    pour chaque attribut de $classe_1$, il existe un attribut de $classe_2$ avec un type compatible,
    pour chaque méthode de $classe_1$, il existe une méthode de $classe_2$ avec des types compatibles,
    il existe une fonction de $tr : classe_1 \rightarrow classe_2$, qui transforme un objet de $classe_1$ en un objet de $classe_2$,
  - chaque association dans l'IDM de *II* doit avoir une contrepartie dans l'IDM de *EI* et les contraintes sur les cardinalités ne sont pas plus fortes que dans l'IDM de *EI*,
  - l'invariant de l'IDM de *EI* implique l'invariant de l'IDM de *II*.
    Cette condition sert à assurer que les états de l'IDM de *II* sont compatibles avec les états de l'IDM de *EI*. Si *EI* requiert une certaine relation entres les objets, *II* doit aussi satisfaire cette relation.

Dans le cas d'Object-Z, les règles de compatibilité sont les mêmes que pour le raffinement de données défini pour Z [Spi92].

3.1.2 Compatibilité entre opérations

Les opérations des interfaces *II* et *EI* sont compatibles si pour chaque opération de *II*, il existe une opération correspondante de *EI* telle que:

- les profils sont compatibles,
- la précondition de l'opération de *II* implique la précondition de l'opération correspondante de *EI*, où les données présentes dans *EI* mais non dans *II* sont remplacées par des valeurs par défaut,
- la postcondition de l'opération de *EI* implique la postcondition de l'opération correspondante de *II*.

Ceci correspond à la notion de raffinement de données.

3.1.3 Compatibilité entre protocoles

Les protocoles des interfaces *II* et *EI* sont compatibles si tous les ordres entre les opérations avec paramètres permis dans *II* sont aussi permis dans *EI*.

3.2 Étude de cas

On veut vérifier qu'il est possible d'utiliser le composant *HotelMgr* avec son interface d'exportation *IHotelMgt* pour satisfaire l'interface d'importation *IHotelHandling* du composant *ReservationSystem*, c'est à dire que *IHotelHandling* $\lhd$ *IHotelMgt*.

Pour cela, il faut montrer la compatibilité entre ces deux interfaces. La correspondance entre les ingrédients des deux interfaces se fait par nom, c'est à dire que chaque ingrédient de l'interface *IHotelHandling* est représenté par l'ingrédient de même nom dans l'interface *IHotelMgt*.

- Dans un premier temps, il faut comparer les modèles de données de ces deux composants décrits dans les figures 3 et 6. On voit que les classes *ReservationDetails* et *HotelDetails* on un attribut en plus, et que la classe *RoomType* n'existe pas dans l'IDM de l'interface *IHotelHandling*.

  Les deux IDM sont compatibles, parce que chaque classe de l'IDM de *IHotelHandling* est présente dans l'IDM de *HotelMgr* et les associations ont les mêmes cardinalités. Pour transformer des objets des classes *ReservationDetails* et *HotelDetails* (fonction *tr*), on rajoute un *roomType* qui s'appelle "Standard" à la classe *ReservationDetails* et la collection avec le seul élément "Standard" comme valeur de l'attribut *roomTypes* de *HotelDetails*.
- L'invariant de *IHotelMgt* implique celui de *IHotelHandling*, parce qu'il a juste la contrainte $hotel = roomType.room.hotel$ en plus.
- Les profils de toutes les opérations sont les mêmes dans les deux interfaces.
  - Les préconditions de l'opération *getHotelDetails* sont les mêmes dans les deux interfaces et la postcondition dans l'interface *IHotelMgt* implique celle de l'interface *IHotelHandling* (elle contient une condition en plus, qui fait référence au type de chambres).
  - La précondition de l'opération *getRoomInfo* de l'interface *IHotelHandling* implique celle de l'interface *IHotelMgt*, parce que la contrainte $hotel.room.roomType.name \rightarrow includes(res.roomType)$ est vraie pour le seul type de chambre disponible "Standard". La postcondition dans l'interface *IHotelMgt* implique celle de l'interface *IHotelHandling* (elle contient une condition en plus, qui fait référence au type de chambres).
  - Pour l'opération *makeReservation*, on a la même situation que pour l'opération *get−RoomInfo*.
- Les protocoles des deux interfaces sont les mêmes.

Ainsi, on a montré $IHotelHandling \lhd IHotelMgt$. Cela signifie que l'on peut utiliser le composant *HotelMgr* permettant de gérer plusieurs types de chambres pour décrire un système de réservation ne disposant que d'un seul type de chambre.

4 Conclusion

Nous avons proposé une manière de spécifier des composants indépendante des langages de spécification connus. Celle-ci est compatible avec les modèles de composants existants. Dans notre approche, nous prenons en compte un certain nombre d'aspects jusqu'alors négligés, à savoir la spécification des interfaces d'importation, les données requises et le protocole d'utilisation du composant. Dans les approches composants connues, on dispose de contrats exprimés uniquement en termes de pré- et de postconditions; les données ne sont pas explicitement prises en compte. Mais elles sont nécessaires pour établir l'interopérabilité entre composants. Celle-ci est couverte dans notre proposition par la notion de compatibilité entre interfaces. La compatibilité est une notion importante pour la définition et l'utilisation de composants car elle permet de garantir l'exploitation de ses avantages, à savoir l'utilisation d'un composant sans en connaître ses détails d'implantation. Cette notion de compatibilité doit être définie le plus indépendamment possible du langage de spécification de composants choisi.

La poursuite de ce travail pourra s'effectuer en décrivant précisément la notion de compatibilité pour différents langages. On s'intéressera aussi à des aspects non fonctionnels qui jouent un rôle important dans l'utilisation de composants ainsi qu'à fournir une méthodologie de construction.

Références

[CAB+94] Coleman (Dereck), Arnold (Patrick), Bodoff (Stéphanie), Dollin (Chris), Gilchrist (Helena), Hayes (F.) et Jeremaes (P.). – *Object-Oriented Development: The Fusion Method*. – Prentice-Hall, 1994.

[CD01] Cheesman (John) et Daniels (John). – *UML Components – A Simple Process for Specifying Component-Based Software*. – Addison-Wesley, 2001.

[HC01] Heineman (George T.) et Councill (William T.). – *Component-Based Software Engineering*. – Addison-Wesley, 2001.

[Hoa85] Hoare (C.A.R.). – *Communicating Sequential Processes*. – Prentice Hall, 1985.

[HSS02] Heisel (Maritta), Santen (Thomas) et Souquières (Jeanine). – Toward a formal model of software components. *In : Proc. 4th International Conference on Formal Engineering Methods*, éd. par George (Chris) et Huaikou (Miao). pp. 57–68. – Springer-Verlag, 2002.

[Mey97] Meyer (Bertrand). – *Object-Oriented Software Construction*. – Prentice-Hall, 1997, second édition.

[Mic95] Microsoft Corporation. – *The Component Object Model Specification, Version 0.9*, 1995. http://www.microsoft.com/com/resources/comdocs.asp.

[Obj98] The Object Mangagement Group (OMG). – *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, février 1998. http://cgi.omg.org/library/corbaiiop.html.

[Ove03] Overhagen (Sven). – Towars a Standardized Specification Framework for Component Development, Discovery, and Configuration. *In : Eighth International Workshop on Component-Oriented Programming, WCOP*, éd. par Bosch (Jan), Szyperski (Clemens) et Beck (Wolfgang). – 2003.

[Reu01] Reussner (Ralf H.). – Adapting Components and Predicting Architectural Properties with Parameterised Contracts. *In : Tagungsband des Arbeitstreffens der GI Fachgruppen 2.1.4 und 2.1.9, Bad Honnef*, éd. par Goerigk (Wolfgang), pp. 33–43. – mai 2001.

[RJB97] Rumbaugh (James), Jacobsen (Ivar) et Booch (Grady). – *Unified Modeling Language Reference Manual*. – Addison-Wesley, 1997.

[Smi99] Smith (Graeme). – *The Object-Z Specification Language*. – Kluwer Academic Publishers, 1999.

[Spi92] Spivey (J. Mike). – *The Z Notation – A Reference Manual*. – Prentice Hall, 1992, 2nd édition.

[Sun97] Sun Microsystems. – *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[Sun01] Sun Microsystems. – *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.

[Szy99] Szyperski (Clemens). – *Component Software*. – ACM Press, Addison-Wesley, 1999.

[WK99] Warmer (Jos) et Kleppe (Anneke G.). – *The Object Constraint Language: Precise Modeling with UML*. – Addison-Wesley, 1999.