Schémas de développement d'adaptateurs à l'aide de B

Arnaud Lanoix, Samuel Colin, Jeanine Souquières

Schémas de développement d'adaptateurs à l'aide de B

Arnaud Lanoix Samuel Colin Jeanine Souquières

LORIA – Nancy Université
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex
{Arnaud.Lanoix,Samuel.Colin,Jeanine.Souquieres}@loria.fr

Résumé

Dans une approche composants pour le développement de logiciels, les composants sont considérés comme des boîtes noires qui communiquent via leurs interfaces. L'interface fournie d'un composant peut être connectée à l'interface requise d'un autre composant si l'interface du premier composant implante les fonctionnalités requises par le second composant. Une description formelle de ces interfaces est nécessaire pour s'assurer de leur compatibilité. En général, les interfaces ne sont pas directement compatibles et un adaptateur doit être introduit. Nous proposons des schémas pour développer des adaptateurs et vérifier l'interopérabilité des composants.

Mots-clés : composant, adaptateur, vérification, construction sûre, raffinement

1 Introduction

L'approche conception de systèmes par assemblage de composants est une approche de développement intéressante et de plus en plus adoptée aujourd'hui [30]. Des composants logiciels "boîte noire" développés par ailleurs sont assemblés les uns avec les autres pour produire le système complet. Le processus d'assemblage sous-jacent est similaire aux méthodes de construction et de réutilisation développées dans d'autres disciplines comme le génie mécanique ou le génie électrique.

Les composants sont assemblés via leurs interfaces. Une interface *fournie* par un composant peut être connectée avec une interface *requise* d'un autre composant si la première offre toutes les fonctionnalités permettant d'implanter la seconde : les composants doivent être connectés de manière appropriée. Afin de garantir cette interopérabilité entre composants, nous considérons chaque connexion entre interfaces fournie et requise de l'architecture et montrons que les interfaces sont compatibles. Une description appropriée des interfaces est primordiale si l'on veut vérifier que l'assemblage est correct.

La spécification formelle des interfaces et la preuve de leur interopérabilité en utilisant la méthode formelle B a été étudiée dans [8, 7, 14]. Grâce à B, nous prouvons que le modèle de l'interface fournie est un *raffinement* correct de l'interface requise ; en d'autres termes, nous prouvons que l'interface fournie correspond à une implantation correcte de l'interface requise et par conséquent, que les composants peuvent être connectés [7].

Dans la plupart des cas, des adaptateurs (ou médiateurs) entre composants, doivent être définis pour assurer l'interopérabilité entre composants. Un adaptateur est un programme qui

réalise la correspondance entre une interface requise et une interface fournie, lorsque celles-ci ne sont pas directement compatibles. Une étude générale de la construction des adaptateurs et de leur preuve en termes du raffinement de l'interface requise incluant le modèle B de l'interface fournie est décrite dans [23, 19]. Cette étude a été étendue avec la prise en compte de modèles d'interfaces différents [10] et de propriétés de sécurité [18].

Dans cet article, nous systématisons notre approche et proposons différents schémas pour développer des adaptateurs et vérifier l'interopérabilité des composants ainsi connectés. Les points forts de notre approche sont :

- l'utilisation de notations simples et de haut niveau pour exprimer l'architecture du système et ses interfaces,
- des schémas d'adaptateurs utilisant les mécanismes classiques de composition et de raffinement,
- des guides pour développer incrémentalement ces adaptateurs,
- la preuve de l'interopérabilité des composants.

L'article est structuré de la manière suivante. Le chapitre 2 présente l'utilisation de la méthode B dans une approche composant. Le chapitre 3 présente la compatibilité directe entre deux interfaces et sa vérification à l'aide de B. Le chapitre 4 présente plusieurs cas d'adaptation de deux interfaces, ainsi que les schémas d'adaptateurs permettant d'exprimer et de vérifier l'interopérabilité. Le chapitre 5 s'intéresse au cas où le nombre de composants est supérieur à deux. Des travaux connexes sont discutés dans le chapitre 6 et une conclusion avec des perspectives d'évolution termine ce papier. L'exemple utilisé tout au long de ce papier est celui du contrôle d'accès à un ensemble de bâtiments.

2 Description de l'approche

Dans l'approche composants que nous proposons [14], l'architecture du système est modélisée à l'aide de diagrammes UML 2.0 [24] annotés par des modèles B associés aux interfaces des différents composants. Les modèles B sont utilisés pour exprimer une spécification formelle des interfaces et ainsi vérifier systématiquement leur compatibilité. Pour cela, nous utilisons deux notions clés de la méthode B [1] :

- le raffinement qui permet un développement incrémental avec préservation de la correction à chaque étape du développement,
- les mécanismes de composition avec les clauses INCLUDES, PROMOTES et EXTENDS.

2.1 Architecture composants

Nous décrivons un système à base de composants à l'aide de plusieurs diagrammes UML :

- les diagrammes de structure composite expriment l'architecture globale du système en termes des composants et des interfaces à connecter ;
- les diagrammes de classes expriment les modèles de données et les signatures des méthodes des interfaces ;
- les PSMs, *Protocol State Machine*, expriment les protocoles d'utilisation pour certaines interfaces. Ces diagrammes ne seront pas utilisés dans ce papier ;
- les diagrammes de séquences permettent d'exprimer certaines interactions possibles entre composants connectés via leurs interfaces.

Les interfaces des composants sont ensuite spécifiées à l'aide de la méthode formelle B, augmentant le degré de confiance dans les systèmes développés : la correction des spécifications

ainsi que la correction du processus de raffinement sont vérifiées à l'aide d'outils [29, 9]. Dans un processus de développement intégré, les modèles B peuvent être obtenus en appliquant des règles systématiques de transformation de UML vers B [22, 20].

2.2 Étude de cas : le contrôle d'accès

Nous illustrons notre propos à l'aide de l'étude de cas du contrôle d'accès à un ensemble de bâtiments [2]. L'objectif est de développer un système chargé de contrôler l'accès de certaines personnes aux différents bâtiments d'un lieu de travail. Le contrôle s'effectue sur la base de l'autorisation que chaque personne concernée possède. Cette autorisation doit lui permettre, sous le contrôle du système, d'entrer dans certains bâtiments et pas dans d'autres. Lorsqu'une personne se trouve à l'intérieur d'un bâtiment, sa sortie doit également être contrôlée par le système afin de savoir à tout instant qui se trouve dans un bâtiment donné.

Chaque personne autorisée dispose d'une carte d'accès avec un code. Des lecteurs de cartes sont installés à chaque entrée et sortie de bâtiment. À proximité de chaque lecteur se trouvent deux voyants, un rouge et un vert, chacun d'eux pouvant être allumé ou éteint. À chaque entrée et sortie de bâtiment se trouve un tourniquet normalement bloqué. Lorsqu'un tourniquet est débloqué par le système, le passage éventuel d'une personne est détecté par un capteur. Chaque tourniquet n'est affecté qu'à une seule tâche, entrer ou sortir.

L'entrée et la sortie obéissent à une procédure systématique :

- si la personne est autorisée à entrer dans le bâtiment concerné (elle est toujours autorisée à sortir), le voyant vert s'allume et le tourniquet se débloque. La spécification originelle fait état d'une contrainte de temps sur la durée de déblocage, contrainte que nous avons préféré abstraire en supposant qu'elle était gérée par le tourniquet lui-même. Dès que la personne franchit le tourniquet le voyant vert s'éteint et le tourniquet se bloque immédiatement. Si la carte n'a pas été reprise au bout d'un certain laps de temps, elle est «avalée» par le lecteur.
- si la personne n'est pas autorisée à entrer dans le bâtiment, le voyant rouge s'allume et le tourniquet reste bloqué. Ici encore le retrait de la carte est soumis à une durée limite, au-delà de laquelle la carte est «avalée» par le lecteur.

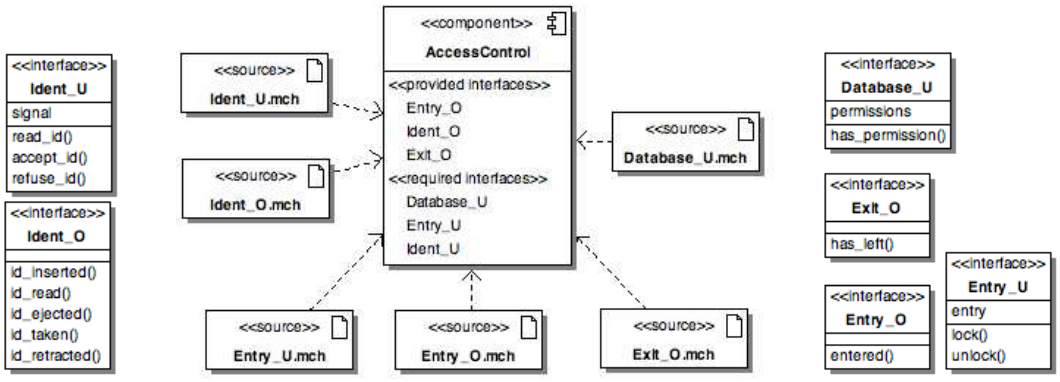


Figure 1. Composant AccessControl

2.2.1 Architecture composants du système de contrôle d'accès

Dans une vue composant, le système de contôle d'accès peut être représenté par AccessControl, donné figure 1. Des interfaces requises et fournies sont exprimées pour répondre aux différents besoins exprimés dans le cahier des charges de ce système :

- les interfaces Ident_O et Ident_U expriment l'ensemble des fonctionnalités liées à l'identification par le contrôleur d'accès. Celui-ci commande le système d'identification par le biais de l'interface Ident_O et reçoit des informations en retour via Ident_U ;
- l'interface Database_U permet au contrôleur d'envoyer des requêtes à une base de données contenant les autorisations des usagers et des informations sur les personnes présentes dans les bâtiments ;
- l'interface Exit_O permet d'informer le contrôleur lorsqu'un usager sort du bâtiment ;
- l'interface Entry_U permet au contrôleur d'accès de commander le blocage/déblocage de l'entrée ; l'interface Entry_O informe le contrôleur du passage d'un usager.

Des modèles B sont associés aux différentes interfaces. Ceux de Entry_U et Database_U sont présentés figures 6 et 8.

Pour répondre aux besoins exprimés par le composant AccessControl, nous disposons des composants suivants, présentés Figure 2.

- Le composant CardReader fournit un pilote de périphérique à un lecteur de cartes. Ses deux interfaces Reader_O et Reader_U correspondent à l'interfaçage entre le lecteur de cartes et son environnement.
- Le composant Light décrit le pilote de commande d'une lampe. L'interface Light_O permet d'allumer et d'éteindre la lampe.
- Le composant Turnstile fournit un pilote chargé de commander un tourniquet. L'interface Turn_O fournit des méthodes pour commander l'ouverture et la fermeture du tourniquet (le modèle B associé est donné figure 6) ; Turn_U propose une méthode pour informer du passage d'un usager.
- Le composant DBNetwork décrit un pilote permettant de connecter une base de données via l'interface DBNet_O. Le modèle B associé est propsé figure 8.

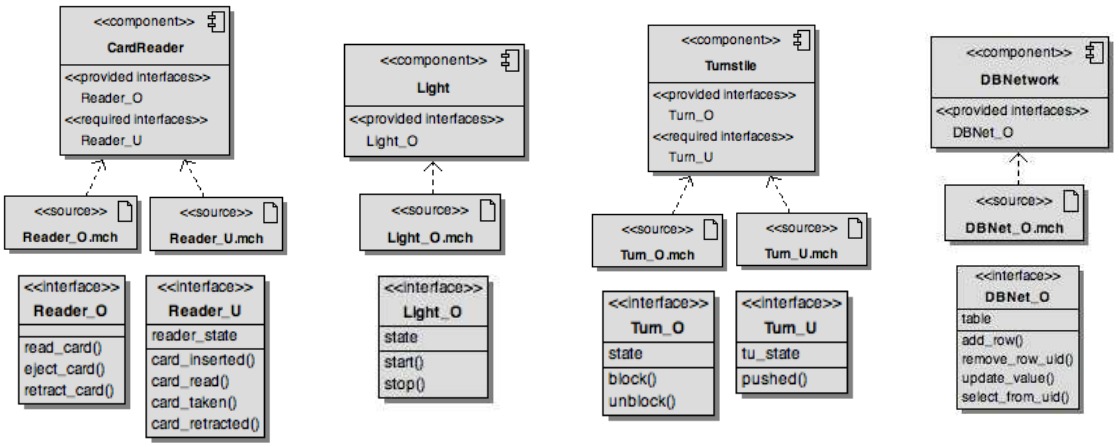


Figure 2. Les composants CardReader, Light, Turnstile et DBNetwork

L'architecture complète du système est décrite Figure 3 sous la forme d'un diagramme de structure composite d'UML. Elle utilise les composants précédemment décrits pour répondre

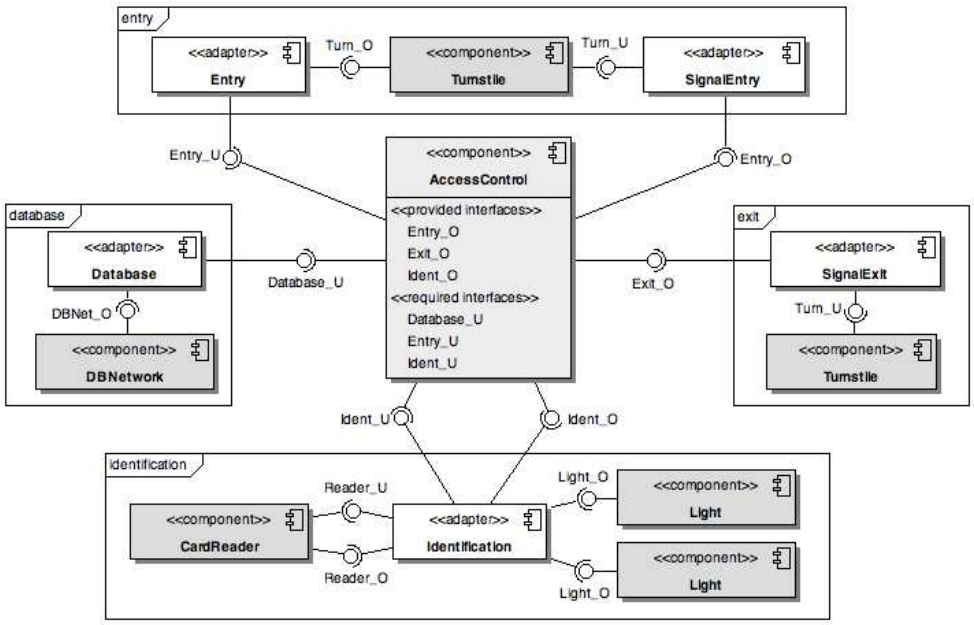


Figure 3. Architecture globale du système de contrôle d'accès

aux besoins exprimés par AccessControl. Comme on peut le remarquer sur cette figure, il est nécessaire de développer des adaptateurs pour connecter ces différents composants. L'objet de ce papier est de proposer des schémas pour exprimer et vérifier des adaptateurs à l'aide de B. Entry, Database et Identification seront détaillés dans les sections suivantes.

3 Compatibilité entre deux interfaces

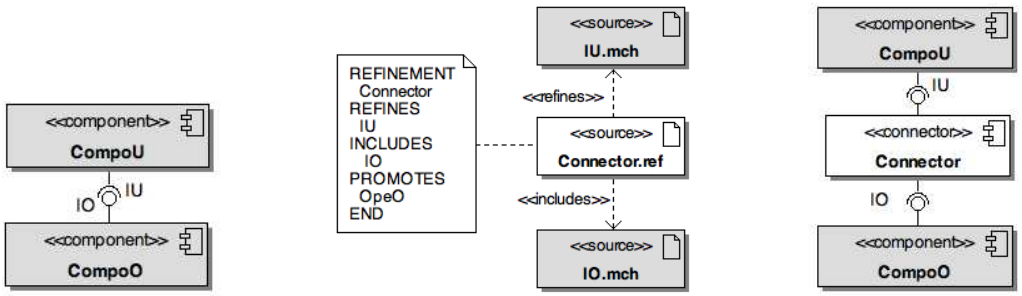
Pour vérifier que deux composants sont *interopérables*, c.à.d. qu'ils peuvent être connectés via leurs interfaces respectives, il faut s'assurer que ces interfaces sont *compatibles*. Plus précisément, il s'agit de montrer que l'interface fournie implante bien les fonctionnalités nécessaires à l'interface requise [7]. Soient CompoU et CompoO deux composants représentés Figure 4(a), tels que :

- CompoU nécessite une interface IU, et
- CompoO implante une autre interface IO.

IO peut fournir plus de fonctionnalités que n'en nécessite IU. A l'aide des mécanismes de composition et de raffinement de B, nous pouvons vérifier la compatibilité entre IU et IO. Nous proposons d'utiliser le schéma de développement donné Figure 4(b) pour construire automatiquement un modèle B, appelé Connector, permettant de démontrer la compatibilité directe entre IU et IO. On prouve que Connector *raffine* le modèle B associé à IU en *incluant* le modèle B associé à IO (clause INCLUDES) et en *promouvant* les opérations OpeO de IO requises par IU (clause PROMOTES).

Dans le cas où IO fournit exactement les méthodes nécessaires à IU, la clause PROMOTES peut être remplacée par la clause EXTENDS.

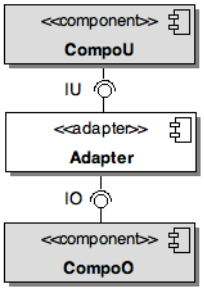
Le modèle B Connector, introduit pour vérifier formellement la compatibilité entre IU et IO, correspond à un *adaptateur* simple qui établit la connexion entre les composants CompoU et CompoO. Il est représenté en UML comme illustré Figure 4(c).



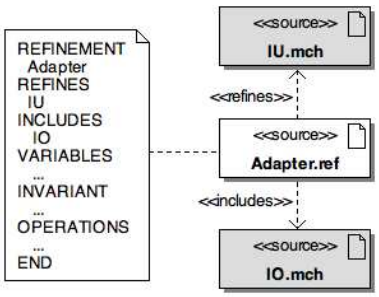
(a) Composants (b) Modèles B (c) Connector

Figure 4. Compatibilité directe entre IU et IO

4 Adaptation entre deux interfaces



(a) Adaptateur



(b) Modèles B

Figure 5. Adaptateur entre IU et IO

La plupart du temps, les interfaces entre deux composants ne sont pas *directement* compatibles et il est nécessaire de développer un adaptateur, c.à.d. un programme qui réalise les fonctionnalités nécessaires à l'interface requise en utilisant l'interface fournie [23].

Examinons le cas où les interfaces IU et IO des composants CompoU et CompoO ne sont pas directement compatibles. Développer un adaptateur consiste principalement à exprimer comment les attributs et les méthodes de l'interface requise IU sont implantés grâce à ceux de IO. Plus précisément,

i) chaque attribut requis par IU doit être exprimé en utilisant les attributs de IO,

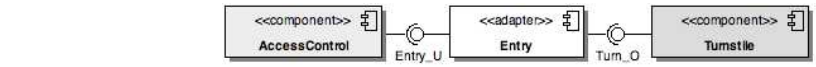
ii) chaque méthode nécessaire à IU doit être exprimée par une combinaison d'appels aux méthodes pertinentes de IO et

iii) les protocoles des interfaces IU et IO doivent être compatibles, c.à.d. que les ordres entre les appels de méthodes permis dans IU doivent aussi être permis dans IO.

L'adaptateur fournit l'interface requise IU tout en requérant l'interface fournie IO comme indiqué Figure 5(a). Tous les adaptateurs suivront ce schéma général, qu'il s'agisse d'appliquer un renommage ou de réaliser des correspondances plus complexes.

Nous proposons d'exprimer l'adaptateur à l'aide d'un raffinement B afin de prouver que l'adaptation est correctement exprimée. La figure 5(b) donne un squelette de l'adaptateur. Il reste bien sûr à compléter les clauses VARIABLES, INVARIANT et OPERATIONS pour respecter les règles *i)*, *ii)* et *iii)*. La preuve du raffinement assurera que le modèle B de l'adaptateur *raffine* le modèle B associé à IU tout en *incluant* correctement le modèle B associé à IO, c.à.d. que l'adaptation est correctement exprimée.

Exemple. Pour connecter AccessControl au composant Turnstile via les interfaces Entry_U et Turn_O, un adaptateur est nécessaire. Le schéma d'adaptation prédédent donne un squelette pour le modèle B de l'adaptateur Entry, comme indiqué Figure 6. Nous complétons ce modèle pour exprimer l'adaptation des éléments de Entry_U en utilisant ceux de Turn_O : ici, il s'agit d'un renommage.




```
MODEL Entry_U
SETS Entry_state = {Locked, Unlocked}
VARIABLES entry
INVARIANT entry ∈ Entry_state
INITIALISATION
  entry := Locked
OPERATIONS

  lock =
  PRE entry = Unlocked
  THEN entry := Locked
  END;

  unlock =
  PRE entry = Locked
  THEN entry := Unlocked
  END

END
```

```
REFINEMENT Entry
REFINES Entry_U
INCLUDES Turn_O
INVARIANT
   (entry = Unlocked ⇒ state = Unblocked) ∧
   (entry = Locked ⇒ state = Blocked)
OPERATIONS

  lock =
  BEGIN
    block
  END;

  unlock =
  BEGIN
    unblock
  END

END
```

```
MODEL Turn_O
SETS State = {Blocked, Unblocked}
VARIABLES state
INVARIANT state ∈ State
INITIALISATION
  state := Blocked
OPERATIONS

  block =
  PRE state = Unblocked
  THEN state := Blocked
  END;

  unblock =
  PRE state = Blocked
  THEN state := Unblocked
  END

END
```

Figure 6. Adaptation simple du tourniquet en entrée

Remarque. Il est à noter que le composant Turnstile est utilisé deux fois, pour l'entrée et pour la sortie d'un bâtiment. D'autres adaptateurs sont donc nécessaires. Les adaptateurs SignalEntry et SignalExit sont similaires à un renommage près. Leur adaptation suit le même processus que celui de l'adaptateur Entry.

4.1 Modèles de données différents

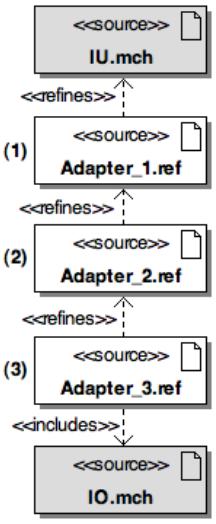


Figure 7.

Il n'est pas toujours facile d'exprimer chaque attribut requis en termes des attributs fournis, surtout si les modèles de données des interfaces IU et IO sont différents. Pour exprimer et vérifier cette correspondance, nous procédons étape par étape, en utilisant le mécanisme de raffinement de B. Un adaptateur peut être exprimé par une série de raffinements successifs commençant, au niveau le plus abstrait, par le modèle B de l'interface requise et se terminant avec l'inclusion du modèle B de l'interface fournie. Dans [10], nous proposons un processus d'adaptation des modèles B en trois étapes de raffinement. Le schéma de l'adaptateur correspondant est détaillé Figure 7.

(1) Adaptation des variables

Il s'agit de préparer la correspondance entre les attributs de IU et ceux de IO :

- de nouvelles variables, qui ré-expriment les variables de IU sont introduites. Elles sont choisies afin de faciliter la mise en correspondance avec celles de IO ;
- le corps de chaque opération de IU est transformé pour prendre en compte ces nouvelles variables.

(2) Adaptation des types de données

Cette étape correspond au transtypage des données :

- les variables introduites à l'étape précédente sont toujours exprimées en termes des types de données de IU. Des fonctions de transtypage sont introduites afin de convertir les types de données de IU vers ceux de IO, et réciproquement. De nouvelles variables sont également introduites par l'application des fonctions de transtypage sur les variables introduites à l'étape précédente ;
- le corps de chaque opération de IU est transformé pour tenir compte des modifications introduites sur les variables.

(3) Inclusion de l'interface fournie

Les deux étapes précédentes ont servi à préparer cette dernière étape qui consiste à inclure le modèle B de l'interface fournie :

- puisque les variables de IU ont été ré-exprimées et que les types de données ont été transformés, il est maintenant facile de mettre en correspondance les variables (modifiées) de IU avec celles de IO ;
- chaque opération de IU est exprimée en termes d'appels aux opérations de IO.

Le processus de développement précédent aide à construire l'adaptateur, mais aussi à réaliser la preuve de l'adaptation. La preuve complète est facilitée par la décomposition en plusieurs étapes. Il est plus facile de démontrer successivement chacune des étapes de l'adaptation plutôt que de démontrer l'ensemble des preuves en une seule étape. Il faut également souligner que les étapes **(1)**, **(2)** et **(3)** ne sont pas toujours toutes nécessaires et qu'il est quelquefois plus facile de subdiviser l'une des étapes en plusieurs raffinements, toujours pour aider la preuve.

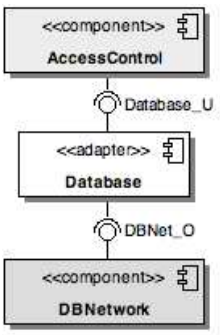


Figure 8.

Exemple. Afin de connecter les interfaces Database_U et DBNet_O, un adaptateur est nécessaire, comme indiqué figure 8. Ces deux interfaces présentent des modèles de données différents. L'interface Database_U permet d'obtenir les portes (bâtiments) autorisées pour un utilisateur donné. L'interface DBNet_O permet de mémoriser dans une base de données des couples (Uid, Value) d'entiers naturels. Plusieurs étapes de raffinement, voir figure 9, sont nécessaires pour réaliser l'adaptation.

(1) Il n'y a pas d'étape d'adaptation des variables, puisque celles-ci peuvent être facilement mises en correspondance : les utilisateurs seront mis en correspondance avec le champ Uid de la base de données, et les portes avec le champ Value.

(2) L'adaptation des types de données est décomposée en deux étapes afin de faciliter la preuve :

- Dans Database_21, une fonction de transtypage user_cast est introduite afin de transformer le domaine de la relation permissions vers les entiers naturels ; une nouvelle variable n_permissions est également introduite.
- Il s'agit maintenant de transformer le codomaine de n_permissions vers les entiers naturels. Une fonction de transtypage, door_cast, ainsi qu'une nouvelle variable, nn_permis--sions, sont introduites dans Database_22.

(3) La dernière étape consiste à associer les champs Uid et Value de DBNet_O à nn_permissions. C'est également lors de cette étape qu'est effectuée la correspondance des méthodes (ici select_from_uid).

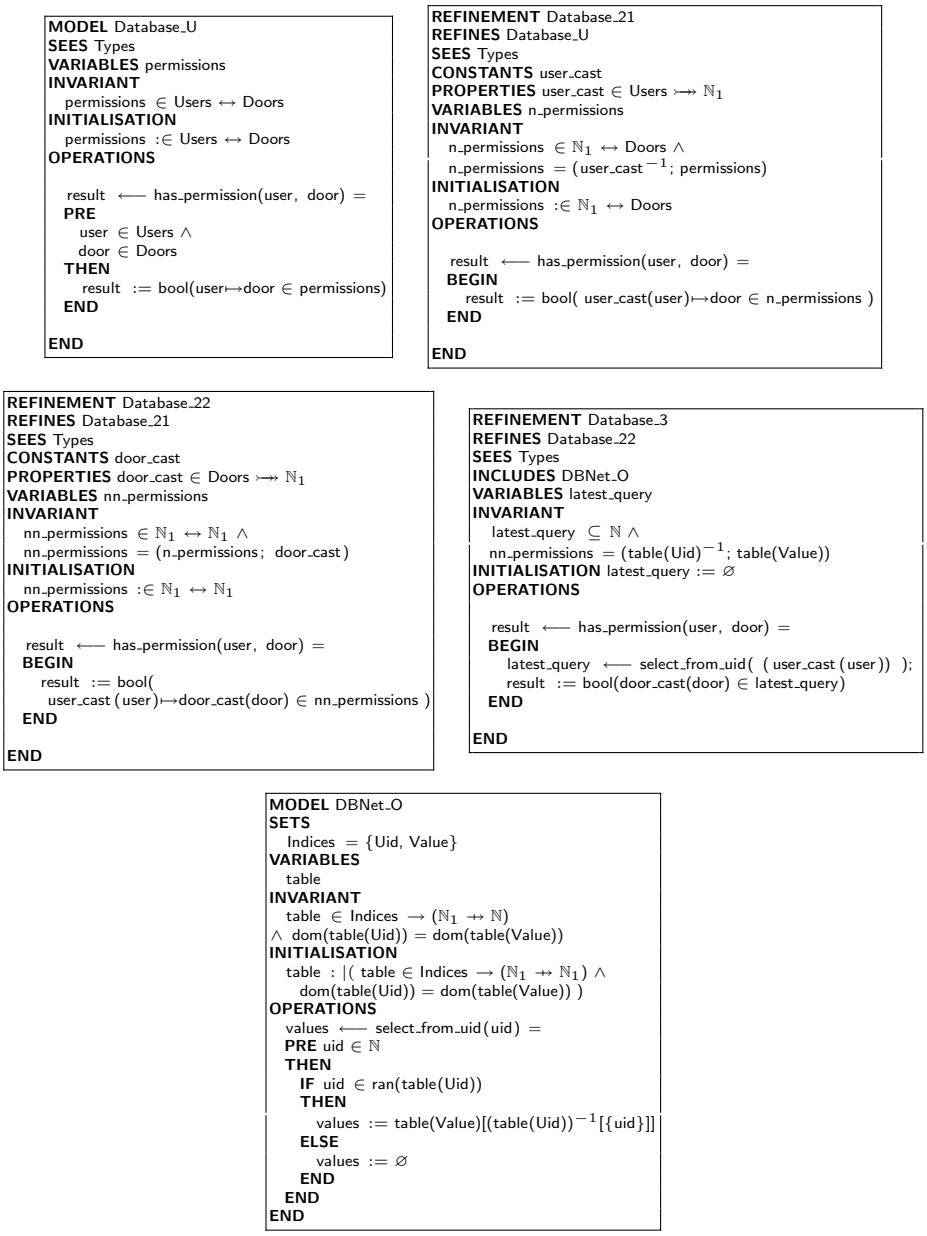


Figure 9. Adaptation des modèles de données

4.2 Protocoles d'appel complexes

Une autre difficulté pour le développement d'un adaptateur correct consiste à établir un protocole d'appel aux méthodes de l'interface fournie pour réaliser les méthodes de l'interface requise. Ce protocole peut être exprimé à l'aide de diagrammes de séquences UML 2.0. Un (ou plusieurs) diagramme de séquences sert à exprimer les appels aux méthodes de IU (fournies par l'adaptateur), puis les appels résultants de l'adaptateur vers les méthodes de IO. La preuve de raffinement du modèle B de l'adaptateur permet de s'assurer que les appels aux méthodes de IO sont valides (preuves d'inclusion) et le raffinement en lui-même assure que l'adaptation est correcte.

5 Généralisation de l'adaptation

La démarche proposée dans la section 4 est généralisée à la connection simultanée de plus de deux interfaces. L'adaptateur réalise les interfaces requises des différents composants à connecter tout en utilisant les interfaces fournies des composants [19] :

- L'adaptateur raffine les modèles B des différentes interfaces requises. Ceci s'exprime en B en introduisant un modèle abstrait intermédiaire qui *étend* les modèles des interfaces requises. Ce modèle est ensuite raffiné par le modèle B de l'adaptateur, Adapter2.ref, comme illustré Figure 10 (dans le cas où deux interfaces sont à réaliser).
- Pour réaliser les différentes interfaces requises, l'adaptateur inclut les modèles B associés aux interfaces fournies des différents composants.

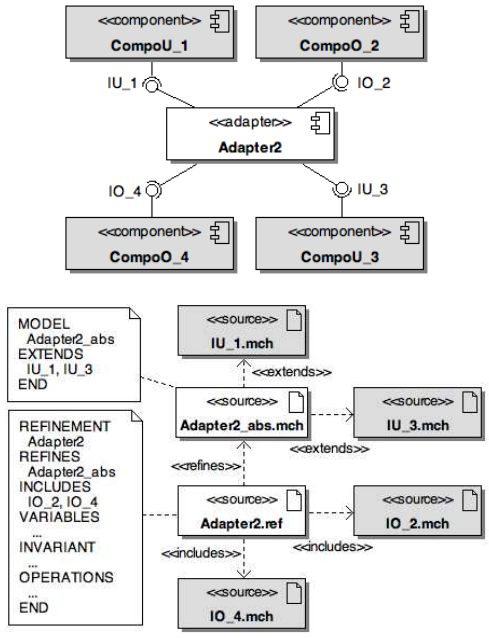


Figure 10. Adaptateur entre interfaces

B propose un mécanisme de renommage associé à la clause INCLUDES, permettant d'utiliser dans un adaptateur, plusieurs instances d'un même composant via des interfaces fournies identiques.

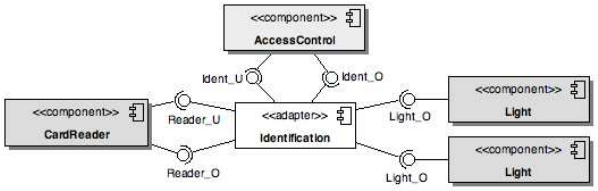
Plusieurs étapes de raffinement peuvent être nécessaires pour faciliter le développement et la preuve de l'adaptateur. Comme plusieurs composants sont en jeu, le protocole d'appels que doit réaliser l'adaptateur peut être complexe et il est souvent plus profitable de commencer par l'exprimer à l'aide d'un diagramme de séquences, mettant en jeu les différentes interfaces des composants à connecter.

Exemple. Pour répondre aux besoins exprimés par Ident_U et Ident_O à propos de l'identification d'un usager vis-à-vis de AccessControl, nous proposons d'utiliser un composant CardReader via ses interfaces Reader_U et Reader_O et deux instances du composant Light via leurs interfaces Light_O. Un adaptateur Identification est nécessaire pour assembler ces différentes interfaces en jeu, comme indiqué figure 11(a). Celui-ci devra expliciter comment implémenter les interfaces requises Ident_U et Reader_U en utilisant les interfaces fournies Ident_O, Reader_O et Light_O.

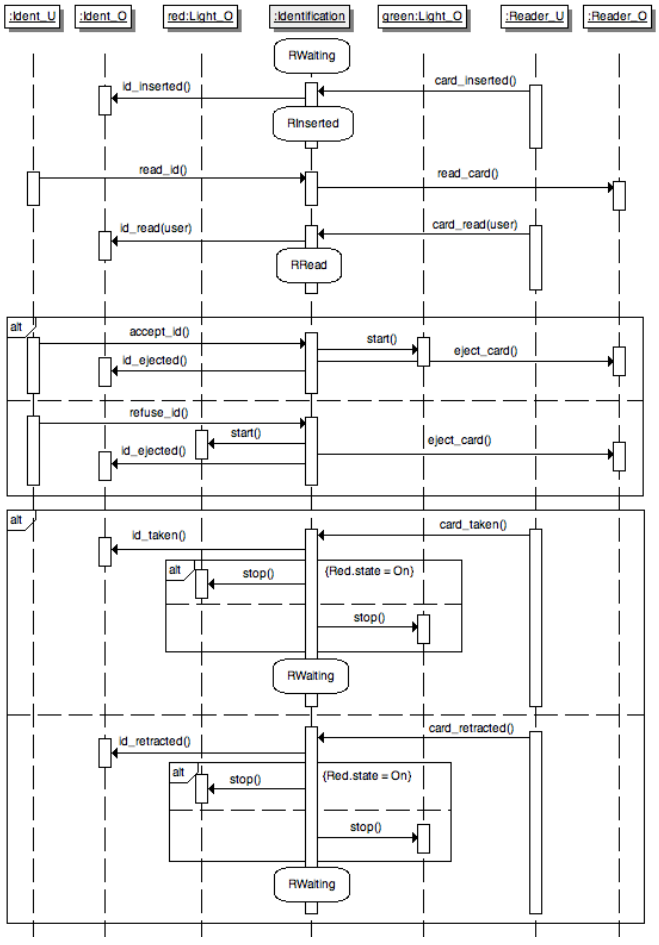
Le diagramme de séquence proposé figure 11(b) permet dans un premier temps d'expliciter le protocole d'appel de l'adaptateur. A chaque méthode des interfaces requises Ident_U et Reader_U, on associe la réaction de l'adaptateur, c.à.d. les appels aux méthodes nécessaires des interfaces fournies. Par exemple, la méthode accept_id() de Ident_U correspond à une notification d'autorisation d'accès d'un usager par le système de contrôle d'accès. L'adaptateur doit réagir en terme des interfaces fournies, en demandant l'allumage d'une lampe verte (Green.start()) afin de confirmer visuellement à l'usager son autorisation d'accès, en éjectant la carte (eject_card()) et en notifiant au contrôleur d'accès l'éjection de la carte (id_ejected()).

Un premier modèle B abstrait nécessaire, Identification_abs, étend les interfaces Reader_U et Ident_U. Le modèle B de l'adaptateur proprement dit est donné figure 11(c). Celui-ci doit raffiner le modèle Identification_abs afin de vérifier que l'adaptateur réalise les différentes interfaces requises, tout en incluant les modèles B des différentes interfaces fournies :

– L'invariant de Identification établit un lien entre les variables à fournir reader_state et signal et les variables fournies par les interfaces Ident_O, Reader_O et Light_O.
– Chacune des méthodes de Reader_U et de Ident_U est reformulée en termes d'appels aux méthodes correspondantes des modèles inclus. La méthode accept_id, par exemple, est réexprimée par la séquence d'appels Green :start ; eject_card ; id_ejected.



(a) Composants UML



(b) Diagramme de séquences

```
REFINEMENT Identification
REFINES Identification_abs
INCLUDES Red.Light_O, Green.Light_O,
         Reader_O,
         Ident_O
VARIABLES reader_state
INVARIANT
  (Green.state = On ⇒ Red.state = Off) ∧
  (Red.state = On ⇒ Green.state = Off) ∧
  ( signal = No_signal ⇒
      (Green.state = Off ∧ Red.state = Off))
INITIALISATION reader_state := RWaiting
OPERATIONS

  card_inserted =
  BEGIN
    id_inserted ;
    reader_state := RInserted
  END;

  read_id =
  BEGIN
    read_card
  END;

  card_read(user) =
  BEGIN
    id_read(user) ;
    reader_state := RRead
  END;

  accept_id =
  BEGIN
    Green.start ;
    eject_card ;
    id_ejected
  END;

  refuse_id =
  BEGIN
    Red.start ;
    eject_card ;
    id_ejected
  END;

  card_taken =
  BEGIN
    id_taken ;
    SELECT Red.state = On THEN Red.stop
    WHEN Green.state = On THEN Green.stop
    END;
    reader_state := RWaiting
  END;

  card_retracted =
  BEGIN
    id_retracted ;
    SELECT Red.state = On THEN Red.stop
    WHEN Green.state = On THEN Green.stop
    END;
    reader_state := RWaiting
  END

END
```

(c) Identification

Figure 11. Adaptateur d'identification

Remarque. Les états des interfaces mises en œuvre restent disjoints, c.à.d. qu'il n'est pas possible de lier ces états, que ce soit au niveau des préconditions des méthodes ou des modifications effectuées, en utilisant les nouveaux états introduits dans le raffinement. Ce phénomène est induit par l'utilisation de composants indépendants. L'adaptateur doit créer des liens qui n'existent pas : ceux-ci imposent un style de programmation défensif, traduit par l'utilisation dans notre exemple de gardes (clause SELECT) plutôt que de préconditions (style offensif).

Les différents adaptateurs présentés ainsi que les interfaces nécessaires ont tous été validés avec B4free. Cela nous permet d'assurer que les adaptations sont correctes et que les différents composants mis en jeu dans l'exemple du système de contrôle d'accès pourront interagir correctement. Le détail des obligations de preuves (OPs) est donné dans le tableau 12.

	OPs évidentes	**OPs**	**OPs** *interactives*
Types	1	0	0
Turn_O	5	0	0
Turn_U	3	0	0
Entry_O	3	0	0
Entry_U	5	0	0
Exit_O	3	0	0
Entry	**9**	**2**	**0**
Signal_Entry	**3**	**0**	**0**
Signal_Exit	**3**	**0**	**0**
DBNet_O	12	10	4
Database_U	3	0	0
Database_21	**6**	**2**	**2**
Database_22	**6**	**2**	**2**
Database_3	**5**	**8**	**2**
Light_O	5	0	0
Reader_O	7	0	0
Reader_U	9	0	0
Ident_O	11	0	0
Ident_U	7	0	0
Identification_abs	**9**	**0**	**0**
Identification	**62**	**6**	**2**
TOTAL	**177**	**30**	**12**

Figure 12. OPs du contrôle d'accès

6 Etat de l'art

Les travaux de recherche relatifs à l'adaptation de composants sont nombreux et la nécessité de disposer de mécanismes d'assemblage performants pour les réaliser a été reconnue dès les années 1990 [5, 15, 16, 11, 6].

Des approches pragmatiques ont porté sur l'analyse des problèmes sous-jacents à l'adaptation de composants existants [13]. Une définition formelle de l'interopérabilité et de l'adaptation de composants a été introduite dans [32]. Dans ce cadre, la spécification du comportement d'un composant est décrite à l'aide de machines à états finis pour lesquelles il existe des techniques et des outils efficaces permettant la vérification de la compatibilité des protocoles.

Reussner et Schmidt considèrent une certaine classe de problèmes dans le contexte des systèmes concurrents [28, 27]. L'incompatibilité des protocoles est résolue par le génération d'adaptateurs en utilisant les interfaces décrites en termes de machines à états finis.

Braccalia & al [4] spécifient un adaptateur comme un ensemble de correspondances entre les méthodes et les paramètres des composants requis et fournis. Un adaptateur est formalisé par un ensemble de propriétés exprimées à l'aide du π-calcul.

Une des premières approches concernant la réutilisation de modules avec adaptation de leurs interfaces est celle proposée par Purtilo et Atlee [26] : ils proposent un langage dédié, Nimble, où l'adaptation entre interfaces requises et fournies est effectuée par le développeur. Notre approche est assez voisine avec l'utilisation de UML et B comme langages, reposant sur des standards et des outils de vérification.

Les travaux présentés dans [25] proposent un processus de génération d'adaptateurs. De nombreux travaux actuels sont dédiés à l'adaptation dynamique [31], qui va plus loin que notre approche : l'adaptation des composants s'effectue lors de l'exécution en recherchant le composant adapté [21, 17]. Malheureusement, ces méthodes reposent sur des besoins forts (relations d'héritage, correspondances à l'exécution des relations entre interfaces, ...) et reposent sur des relations de sous-types.

Le papier [12] présente un cadre pour modéliser des architectures composants en utilisant des techniques formelles (réseaux de Petri et CSP) : les connexions entre interfaces requises et fournies sont représentées par des transformations de graphes utilisant des notions de composition, d'extension et de raffinement. Notre approche est similaire avec l'utilisation de B pour exprimer les transformations comme des raffinements entre interfaces requises et fournies.

Zaremski et Wing [33] proposent une approche intéressante pour comparer deux composants logiciels, permettant de décider si un composant peut être remplacé par un autre. Ils utilisent les spécifications algébriques pour modéliser le comportement des composants et le prouveur Larch pour prouver la correspondance entre composants.

La génération automatique d'adaptateurs est limitée à une certaine classe de problèmes car la vérification de l'interopérabilité repose sur la décidabilité de l'inclusion des composants. Dans notre approche, nous proposons des schémas pour construire et vérifier les adaptateurs, en fonction de différents cas de figures de l'architecture, sans aller jusqu'à leur génération automatique.

7 Conclusion

L'approche composants est un paradigme bien connu et utilisé dans le développement de logiciels, aussi bien dans le milieu académique que dans le milieu industriel. Dans cette approche, les composants sont considérés comme des boîtes noires décrites en termes de leur comportement visible et de leurs interfaces, qu'elles soient requises ou fournies. Des adaptateurs doivent être définis pour construire un système à l'aide de composants. Un adaptateur est un programme qui définit comment les interfaces requises sont réalisées en termes des interfaces fournies : il exprime la correspondance entre variables, types et opérations. Nous proposons une approche formelle pour développer ces adaptateurs avec des schémas pour les construire et les vérifier, en fonction des différents cas de figures de l'architecture. Nous ne proposons pas de les générer automatiquement.

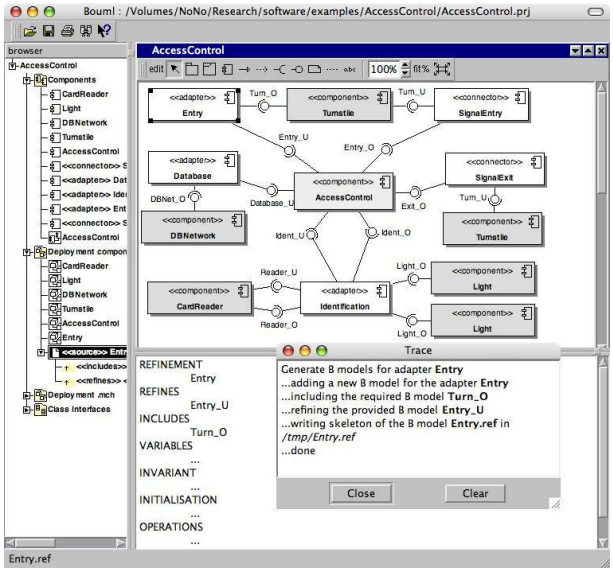


Figure 13. BOUML pour générer un modèle B

Grâce à l'utilisation de la méthode B, de ses mécanismes d'assemblage et de raffinement pour modéliser les interfaces et les adaptateurs, nous obtenons la preuve de l'interopérabilité entre les différents composants. Le prouveur B garantit que l'adaptateur est une implantation correcte des fonctionnalités attendues en termes des composants existants. La vérification de l'interopérabilité entre les composants connectés est effectuée aux niveaux signature, sémantique et protocole.

L'implantation d'un plugin pour BOUML [3] fondé sur les schémas de développement présenté dans cet article est en cours : la figure 13 montre la génération du squelette du

modèle B correspondant à l'adaptateur Entry.

L'extension de l'approche avec la prise en compte de propriétés de sécurité dans une architecture composants existante, sans modification de ses fonctionnalités de base [18] est en cours d'étude. Ce travail doit également être complété par un outil d'aide à la détection des incompatibilités.

Références

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] Afadl2000. Etude de cas : système de contrôle d'accès. In *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. actes LSR/IMAG.

[3] BOUML. UML2 tool box BOUML, release 2.21.2, Jan 2007. http://bouml.free.fr.

[4] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. In *Journal of Systems and Software*, volume 74, pages 45–54. Elsevier Science Inc., 2005.

[5] A. W. Brown and K. C. Wallnan. Engineering of component-based systems. In *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, page 414. IEEE Computer Society, 1996.

[6] C. Canal, J. M. Murillo, and P Poizat. Software adaptation. *L'Objet*, 12(1) :9–31, 2006.

[7] S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 160 :157–172, 2006.

[8] S. Chouali and J. Souquières. Verifying the compatibility of component interfaces using the B formal method. In *International Conference on Software Engineering Research and Practice (SERP'05)*, pages 850–856. CSREA Press, 2005.

[9] Clearsy. B4free, 2004. http://www.b4free.com.

[10] S. Colin, A. Lanoix, and J. Souquières. Trustworthy interface compliancy : data model adaptation. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Satellite workshop of ETAPS*. Electronic Notes in Theoretical Computer Science (ENTCS), 2007. to be published.

[11] I. Crnkovic, S. Larsson, and M. Chaudron. Component-based development process and component lifecycle. In *27th International Conference Information Technology Interfaces (ITI)*. IEEE, 2005.

[12] H. Ehrig, J. Padberg, B. Braatz, M. Klein, F. Orejas, S. Perez, and E. Pino. A generic framework for connector architectures based on components and transformation. In *FESCA'04, satellite of ETAPS'04*, volume 108, pages 53–67. ENTCS, 2004.

[13] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch : Why Reuse is so Hard. *IEEE Software*, 12(6) :17–26, 1999.

[14] D. Hatebur, M. Heisel, and J. Souquières. A Method for Component-Based Software and System Development. In *Proceedings of the 32tnd Euromicro Conference on Software Engineering And Advanced Applications*, pages 72–80. IEEE Computer Society, 2006.

[15] G. Heineman and H. Ohlenbusch. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.

[16] M. Heisel, T. Santen, and J. Souquières. Toward a formal model of software components. In *Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02*, number 2495 in LNCS, pages 57–68. Springer-Verlag, 2002.

[17] G. Kniesel. Type-safe delegation for run-time component adaptation. *Lecture Notes in Computer Science*, 1628 :351–366, 1999.

[18] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquières. Enhancing Dependability of Component-based Systems. In *Reliable Software Technologies Ada-Europe 2007*, LNCS. Springer, 2007. to be published.

[19] A. Lanoix and J. Souquières. A Trustworthy Assembly of COTS Components. *e-Informatica Software Engineering Journal (ISEJ)*, 2007. to be published.

[20] H. Ledang and J. Souquières. Modeling class operations in B : application to UML behavioral diagrams. In *ASE'2001 : 16th IEEE International Conference on Automated Software Engineering*, pages 289–296. IEEE Computer Society, 2001.

[21] K.-U. Mätzel and P. Schnorf. Dynamic component adaptation. Technical report, Ubilab laboratory, Union Bank of Switzerland, Zürich, Switzerland, June 1997.

[22] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proceedings of the Formal Method Conference*, number 1708 in LNCS, pages 875–895. Springer-Verlag, 1999.

[23] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation : Specification and Verification. In *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP'06)*, pages 23–30, 2006.

[24] Object Management Group (OMG). *UML Superstructure Specification*, 2005. version 2.0.

[25] P. Poizat, G. Salaün, and M. Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. In *FACS'06*. Electronic Notes in Theoretical Computer Science, 2006. to appear.

[26] J.M. Purtilo and J.M. Atlee. Module reuse by interface adaptation. *Software - Practice and Experience*, 21(6) :539–556, 1991.

[27] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality : Methods and Techniques*, pages 287–325. 2003.

[28] H. W. Schmidt and R. H. Reussner. Generating adapters fo concurrent component protocol synchronisation. In I. Crnkovic, S. Larsson, and J. Stafford, editors, *Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 213–229, 2002.

[29] Steria – Technologies de l'information. *Obligations de preuve : Manuel de référence, version 3.0*, 1998.

[30] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.

[31] WCAT2006. Coordination and Adaptation Techniques : Bridging the Gap Between Design and Implementation. In S. Becker, C. Canal, N. Diakov, J.-M. Murillo, P. Poizat, and M. Tivoli, editors, *Proceedings of the Third International Workshop on Coordination and A daptation Techniques for Software Entities*, 2006.

[32] D. D M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2) :292–333, 1997.

[33] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4) :333–369, 1997.