

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

Arnaud Lanoix, Samuel Colin et 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 [1]. 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 [2, 3]. 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 [2].

Dans la plupart des cas, des adaptateurs (ou médiateurs) entre composants, doivent être définis pour assurer l'interopérabilité. 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 [4, 5]. Cette étude a été étendue avec la prise en compte de modèles d'interfaces différents [6] et de propriétés de sécurité [7].

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. La section 2 présente l'utilisation de la méthode B dans une approche composant. La section 3 présente la compatibilité directe entre deux interfaces et sa vérification à l'aide de B. La section 4 présente plusieurs cas d'adaptation de deux interfaces, ainsi que les schémas d'adaptateurs permettant d'exprimer et de vérifier l'interopérabilité. La section 5 s'intéresse au cas où le nombre de composants est supérieur à deux. Des travaux connexes sont discutés dans la section 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 [3], l'architecture du système est modélisée à l'aide de diagrammes UML 2.0 [8] 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.

### 2.1 La méthode B

La méthode B [9] est une méthode formelle basée sur la théorie des ensembles, permettant un développement incrémental grâce au raffinement. Un développement commence avec la définition d'une spécification abstraite qui est ensuite raffinée pas à pas jusqu'à l'obtention d'une implantation. Cette méthode a été appliquée avec succès dans le développement d'applications réelles complexes, comme le projet METEOR [10] ou le métro val [11]. Elle est supportée

par des outils robustes. Des obligations de preuves pour la consistance des invariants et la préservation du raffinement sont générées automatiquement par les outils [12, 13].

- Dans notre approche, nous utilisons deux notions clés de la méthode B [9] :
- 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**.

## 2.2 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. Dans un processus de développement intégré, les modèles B peuvent être obtenus en appliquant des règles systématiques de transformation de UML vers B [14, 15].

## 2.3 Étude de cas : le contrôle d'accès

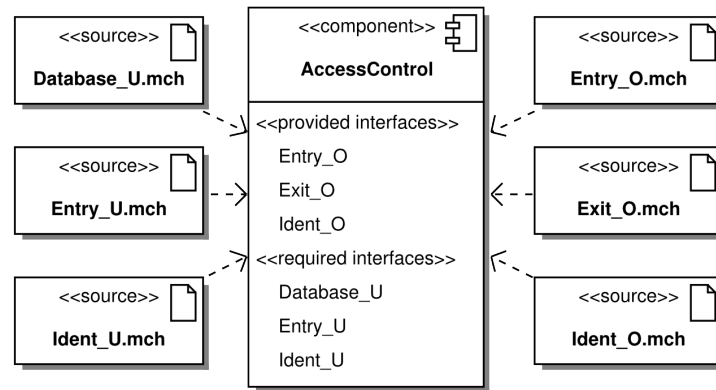
Nous illustrons notre propos à l'aide de l'étude de cas du contrôle d'accès à un ensemble de bâtiments [16]. L'objectif est de développer un système chargé de contrôler l'accès de certaines personnes aux différents bâtiments d'un lieu de travail. Le contrôle s'effectue sur la base de l'autorisation que 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 au protocole suivant :

- 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 débloquent. 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.



**Fig. 1.** Composant AccessControl

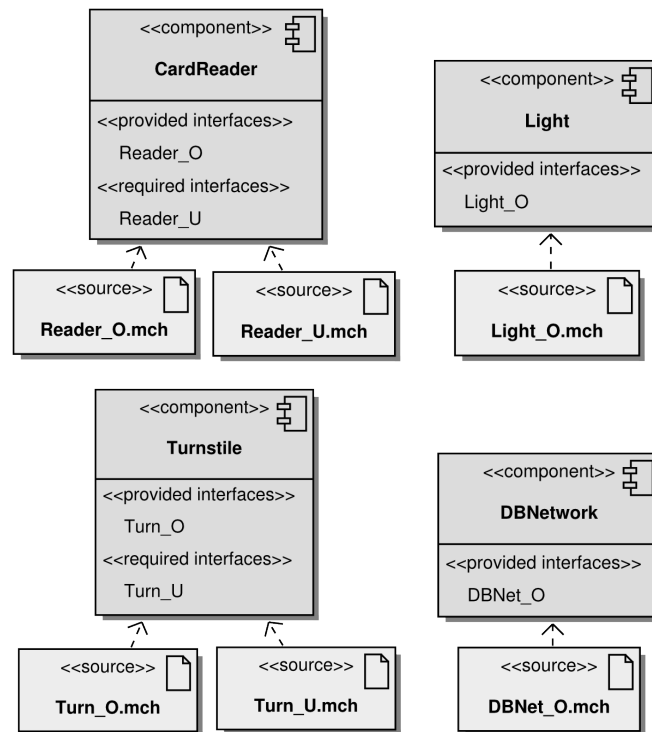
Dans une vue composant, le système de contrôle d'accès peut être représenté par **AccessControl**, présenté figure 1. Des interfaces requises et fournies sont introduites pour répondre aux différents besoins exprimés dans le cahier des charges de ce système (nous indiquons les noms des méthodes de ces interfaces entre parenthèses) :

- les interfaces **Ident\_O** et **Ident\_U** proposent 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** (**id\_inserted**, **id\_read**, **id\_ejected**, **id\_taken**, **id\_retracted**) et reçoit des informations en retour via **Ident\_U** (**read\_id**, **accept\_id**, **refuse\_id**) ;

- l'interface **Database\_U** (**has\_permission**) 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** (**has\_left**) permet d'informer le contrôleur lorsqu'un usager sort du bâtiment ;
- l'interface **Entry\_U** (**lock**, **unlock**) permet au contrôleur d'accès de commander le blocage/déblocage de l'entrée ; l'interface **Entry\_O** (**entered**) 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 8 et 10.

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

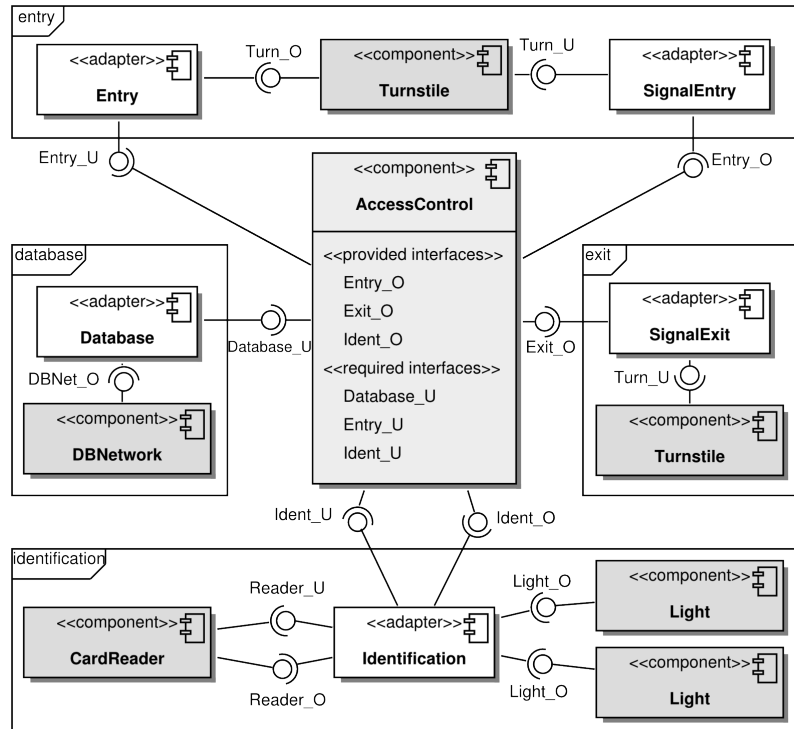


**Fig. 2.** Les composants1 CardReader, Light, Turnstile et DBNetwork

- Le composant **CardReader** fournit un pilote de périphérique à un lecteur de cartes. Ses deux interfaces **Reader\_O** (**read\_card**, **eject\_card**, **retract\_card**) et **Reader\_U** (**card\_inserted**, **card\_read**, **card\_taken**, **card\_retracted**) 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** (**start**,**stop**) permet d'allumer et d'éteindre la lampe.
- Le composant **Turnstile** fournit un pilote chargé de commander un tourniquet. L'interface **Turn\_O** (**block**, **unblock**) fournit des méthodes pour commander l'ouverture et la fermeture du tourniquet (le modèle B associé est donné figure 8); **Turn\_U** (**pushed**) 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** (**add\_row**, **remove\_row\_uid**, **update\_value**, **select\_from\_uid**). Le modèle B associé est proposé figure 10.

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 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.



**Fig. 3.** Architecture globale du système de contrôle d'accès

### 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*.

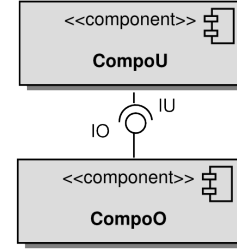
Plus précisément, il s'agit de montrer que l'interface fournie implante bien les fonctionnalités nécessaires à l'interface requise [2]. Soient **CompoU** et **CompoO** deux composants représentés Figure 4, tels que :

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

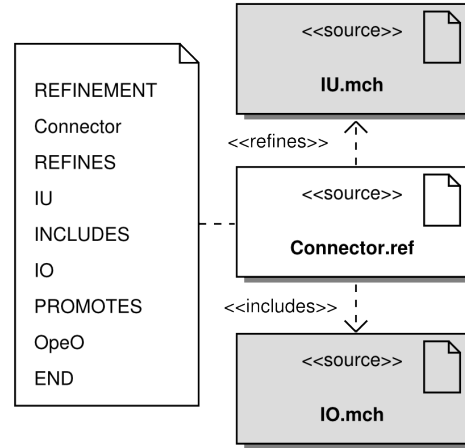
IO peut fournir plus de fonctionnalités que n'en nécessite IU. A l'aide des mécanismes de composition et de raffinement de B, nous pouvons vérifier la compatibilité entre IU et IO.

Nous proposons d'utiliser le schéma de développement donné Figure 5 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).

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**.



**Fig. 4.** Compatibilité entre IU et IO (UML)



**Fig. 5.** Compatibilité entre IU et IO (B)

### 4 Adaptation entre deux interfaces

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 [4].

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,

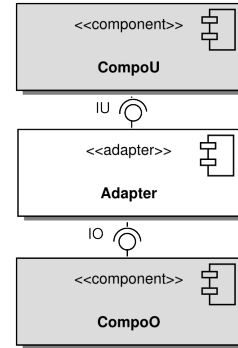
- i) chaque attribut requis par IU doit être exprimé en utilisant les attributs de IO,
- ii) chaque méthode nécessaire à IU doit être exprimée par une combinaison d'appels aux méthodes pertinentes de IO et
- iii) les protocoles des interfaces IU et IO doivent être compatibles, c.à.d. que les ordres entre 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 6. Tous les adaptateurs suivront ce schéma général, qu'il s'agisse d'appliquer un renommage ou de réaliser des correspondances plus complexes.

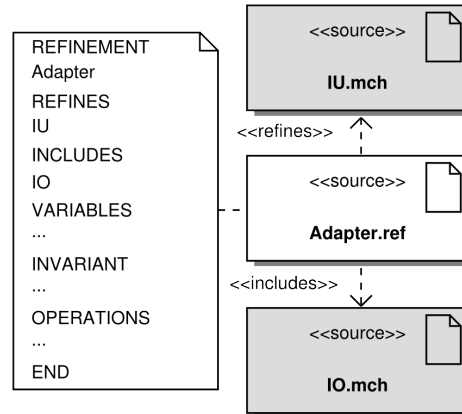
Nous proposons d'exprimer, à l'aide d'un raffinement B, l'adaptateur afin de prouver que l'adaptation est correctement exprimée. La figure 7 donne un squelette de l'adaptateur. Il reste bien sûr à compléter les clauses **VARIABLES**, **INVARIANT** et **OPERATIONS** pour respecter les règles i), ii) et iii) énoncées ci-dessus. La preuve du raffinement assurera que le modèle B de l'adaptateur *refine* 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 Access-Control au composant Turnstile via les interfaces Entry\_U et Turn\_O, un adaptateur est nécessaire. Le schéma d'adaptation précédent donne un squelette pour le modèle B de l'adaptateur Entry, comme indiqué Figure 8. 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.

*Remarque.* Il est à noter que le composant Turnstile est utilisé deux fois dans l'application, 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.

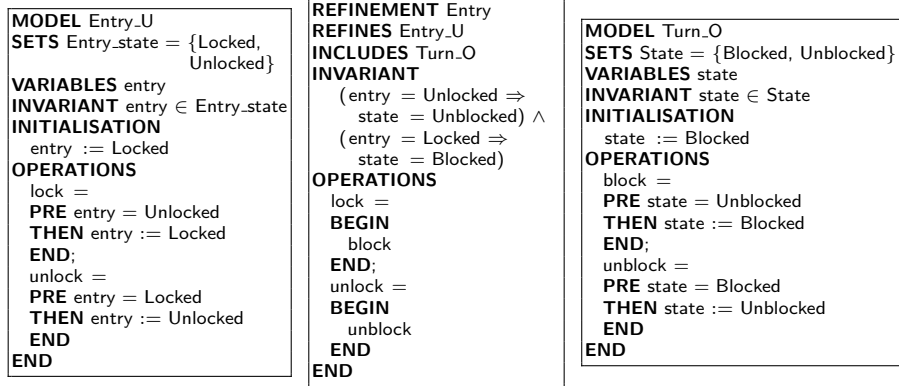


**Fig. 6.** Adaptateur entre IU et IO (UML)



**Fig. 7.** Adaptateur entre IU et IO (B)





**Fig. 8.** Adaptation simple du tourniquet en entrée

#### 4.1 Modèles de données différents

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 [6], nous proposons un processus d'adaptation des modèles B en trois étapes de raffinement. Le schéma de l'adaptateur correspondant est détaillé Figure 9.

##### (1) Adaptation des variables

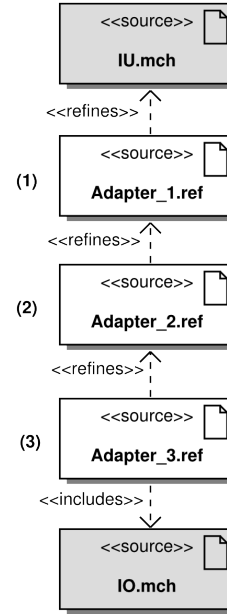
Il s'agit de préparer la correspondance entre les attributs de IU et ceux de IO :

- 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,



**Fig. 9.**

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 faciliter la preuve.

*Exemple.* Afin de connecter les interfaces **Database\_U** et **DBNet\_O**, un adaptateur est nécessaire. 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 10, 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 sont 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** en le domaine des entiers naturels ; une nouvelle variable **n\_permissions** est également introduite.
  - Il s'agit maintenant de transformer le codomaine de **n\_permissions** en le domaine des entiers naturels. Une fonction de transtypage, **door\_cast**, ainsi qu'une nouvelle variable, **nn\_permissions**, 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-à-dire indiquer quelles sont les relations entre les

```

MODEL Database_U
SEES Types
VARIABLES permissions
INVARIANT
  permissions  $\in \text{Users} \leftrightarrow \text{Doors}$ 
INITIALISATION
  permissions :=  $\in \text{Users} \leftrightarrow \text{Doors}$ 
OPERATIONS
  result  $\leftarrow \text{has\_permission}(\text{user}, \text{door}) =$ 
  PRE
    user  $\in \text{Users} \wedge$ 
    door  $\in \text{Doors}$ 
  THEN
    result :=  $\text{bool}(\text{user} \mapsto \text{door} \in \text{permissions})$ 
  END
END

```

```

REFINEMENT Database_21
REFINES Database_U
SEES Types
CONSTANTS user_cast
PROPERTIES user_cast  $\in \text{Users} \mapsto \mathbb{N}_1$ 
VARIABLES n_permissions
INVARIANT
  n_permissions  $\in \mathbb{N}_1 \leftrightarrow \text{Doors} \wedge$ 
  n_permissions =  $(\text{user\_cast}^{-1}; \text{permissions})$ 
INITIALISATION
  n_permissions :=  $\in \mathbb{N}_1 \leftrightarrow \text{Doors}$ 
OPERATIONS
  result  $\leftarrow \text{has\_permission}(\text{user}, \text{door}) =$ 
  BEGIN
    result :=  $\text{bool}(\text{user\_cast}(\text{user}) \mapsto \text{door} \in \text{n\_permissions})$ 
  END
END

```

```

REFINEMENT Database_22
REFINES Database_21
SEES Types
CONSTANTS door_cast
PROPERTIES door_cast  $\in \text{Doors} \mapsto \mathbb{N}_1$ 
VARIABLES nn_permissions
INVARIANT
  nn_permissions  $\in \mathbb{N}_1 \leftrightarrow \mathbb{N}_1 \wedge$ 
  nn_permissions =  $(\text{n\_permissions}; \text{door\_cast})$ 
INITIALISATION
  nn_permissions :=  $\in \mathbb{N}_1 \leftrightarrow \mathbb{N}_1$ 
OPERATIONS
  result  $\leftarrow \text{has\_permission}(\text{user}, \text{door}) =$ 
  BEGIN
    result :=  $\text{bool}(\text{user\_cast}(\text{user}) \mapsto \text{door\_cast}(\text{door}) \in \text{nn\_permissions})$ 
  END
END

```

```

REFINEMENT Database_3
REFINES Database_22
SEES Types
INCLUDES DBNet_O
VARIABLES latest_query
INVARIANT
  latest_query  $\subseteq \mathbb{N} \wedge$ 
  nn_permissions =  $(\text{table}(\text{Uid})^{-1}; \text{table}(\text{Value}))$ 
INITIALISATION latest_query :=  $\emptyset$ 
OPERATIONS
  result  $\leftarrow \text{has\_permission}(\text{user}, \text{door}) =$ 
  BEGIN
    latest_query  $\leftarrow$ 
      select_from_uid (  $\text{user\_cast}(\text{user})$  );
    result :=  $\text{bool}(\text{door\_cast}(\text{door}) \in \text{latest\_query})$ 
  END
END

```

```

MODEL DBNet_O
SETS
  Indices =  $\{\text{Uid}, \text{Value}\}$ 
VARIABLES
  table
INVARIANT
  table  $\in \text{Indices} \rightarrow (\mathbb{N}_1 \mapsto \mathbb{N})$ 
   $\wedge \text{dom}(\text{table}(\text{Uid})) = \text{dom}(\text{table}(\text{Value}))$ 
INITIALISATION
  table :  $\mid (\text{table} \in \text{Indices} \rightarrow (\mathbb{N}_1 \mapsto \mathbb{N}_1) \wedge$ 
     $\text{dom}(\text{table}(\text{Uid})) = \text{dom}(\text{table}(\text{Value})) )$ 
OPERATIONS
  values  $\leftarrow \text{select\_from\_uid}(\text{uid}) =$ 
  PRE uid  $\in \mathbb{N}$ 
  THEN
    IF uid  $\in \text{ran}(\text{table}(\text{Uid}))$ 
    THEN
      values :=  $\text{table}(\text{Value})[(\text{table}(\text{Uid}))^{-1}\{\text{uid}\}]$ 
    ELSE
      values :=  $\emptyset$ 
    END
  END
END

```

**Fig. 10.** Adaptation des modèles de données

structures de données. C'est également lors de cette étape que le corps des méthodes se réduit à l'appel des méthodes correspondantes du composant fourni (ici, `has_permission` appelle `select_from_uid`).

## 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 [5].

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 11 (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.

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

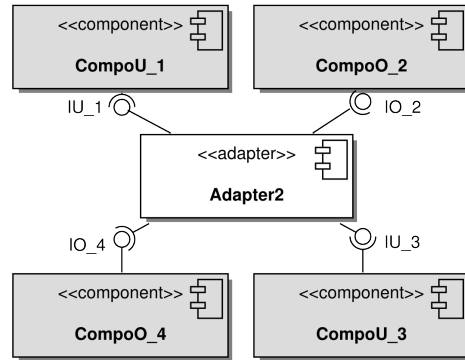
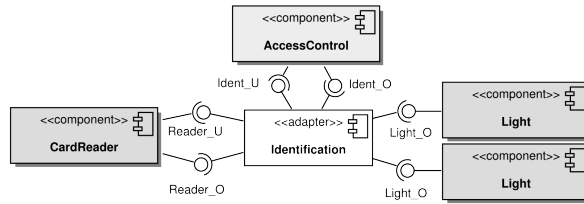


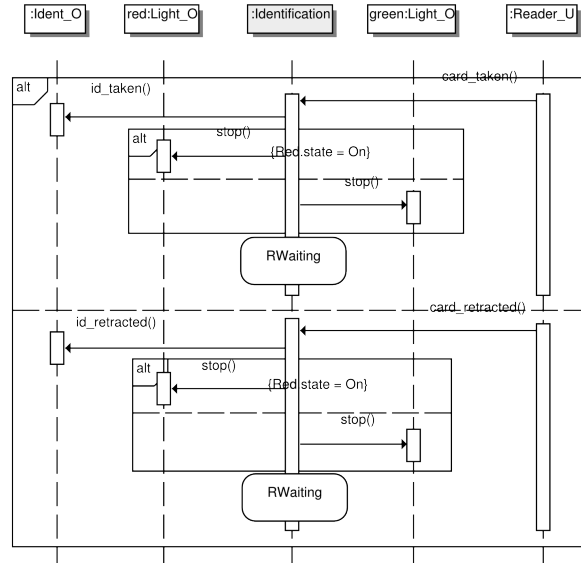
Fig. 11. Adaptateur entre interfaces

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 son interface `Light_O`. Un adaptateur `Identification` est nécessaire pour assembler ces différentes interfaces en jeu, comme indiqué figure 12(a). Celui-ci doit expliciter comment implanter les interfaces requises `Ident_U` et `Reader_U` en utilisant les interfaces fournies `Ident_O`, `Reader_O` et `Light_O`.



(a) Composants UML



(b) Diagramme de séquences (extrait)

```

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_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

**Fig. 12.** Adaptateur d'identification

Le diagramme de séquences proposé figure 12(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 termes des interfaces fournies, en demandant l'allumage d'une lampe verte (`Green.start()`) afin de confirmer visuellement à l'utilisateur 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 12(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`.

*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 1.

Modèles B\OPs	évidentes	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
<b>TOTAL</b>	<b>177</b>	<b>30</b>	<b>12</b>

**Tab. 1.** Obligations de preuves

## 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 [17, 18, 19, 20].

Une des premières approches concernant la réutilisation de modules avec adaptation de leurs interfaces est celle proposée par Purtilo et Atlee [21] : 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.

Des approches pragmatiques ont porté sur l'analyse des problèmes sous-jacents à l'adaptation de composants existants. Une définition formelle de l'interopérabilité et de l'adaptation de composants a été introduite dans [22]. 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.

Zaremski et Wing [23] 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.

Reussner et Schmidt considèrent une certaine classe de problèmes dans le contexte des systèmes concurrents [24, 25]. L'incompatibilité des protocoles est résolue par la génération d'adaptateurs en utilisant les interfaces décrites en termes de machines à états finis.

Les travaux présentés dans [26] proposent un processus de génération d'adaptateurs. De nombreux travaux actuels sont dédiés à l'adaptation dynamique [27], qui va plus loin que notre approche : l'adaptation des composants s'effectue lors de l'exécution en recherchant le composant adapté [28, 29]. Ces méthodes se basent sur l'hypothèse de l'existence de relations d'héritages (avec une possible transitivité) entre une interface fournie et une classe qu'on sait pouvoir utiliser. Elles sont donc fortement basées sur la notion de (sous-)typage dans un contexte de programmation objet, et donc sont moins flexibles en termes d'expressivité que notre approche, bien qu'elles apportent l'adaptation dans un contexte dynamique.

Le papier [30] 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.

Braccalia & al [31] 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  $\pi$ -calcul.

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.

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 implémentation 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'implémentation d'un plugin pour BOUML<sup>1</sup> fondé sur les schémas de développement présentés 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 fonction-

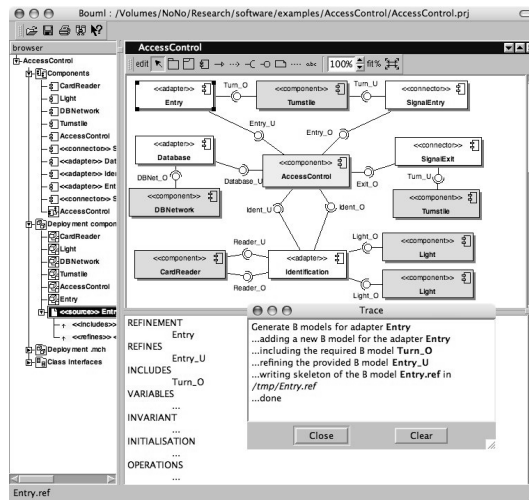


Fig. 13. BOUML pour générer un modèle B

<sup>1</sup> <http://bouml.free.fr>



nalités de base [7] est en cours d'étude. Ce travail doit également être complété par un outil d'aide à la détection des incompatibilités.

## Références

- [1] Szyperski, C. : Component Software. ACM Press, Addison-Wesley (1999)
- [2] Chouali, S., Heisel, M., Souquières, J. : Proving component interoperability with B refinement. *Electronic Notes in Theoretical Computer Science (ENTCS)* **160** (2006) 157–172
- [3] Hatebur, D., Heisel, M., Souquières, J. : A method for component-based software and system development. In : *Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced Applications*, IEEE Computer Society (2006) 72–80
- [4] Mouakher, I., Lanoix, A., Souquières, J. : Component adaptation : Specification and verification. In : *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP'06)*, satellite workshop of ECOOP. (2006) 23–30
- [5] Lanoix, A., Souquières, J. : A trustworthy assembly of components using the B refinement. *e-Informatica Software Engineering Journal (ISEJ)* **2**(1) (2008) 19 pages, published in advances of print.
- [6] Colin, S., Lanoix, A., Souquières, J. : Trustworthy interface compliancy : data model adaptation. In : *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA)*, Satellite workshop of ETAPS. (March 2007) 13 pages. To be published in *Electronic Notes in Theoretical Computer Science (ENTCS)*.
- [7] Lanoix, A., Hatebur, D., Heisel, M., Souquières, J. : Enhancing dependability of component-based systems. In Verlag, S., ed. : *Reliable Software Technologies Ada-Europe 2007*. Number 4498 in LNCS, Springer Verlag (2007) 41–54
- [8] Object Management Group (OMG) : UML Superstructure Specification. (2005) version 2.0.
- [9] Abrial, J.R. : The B Book. Cambridge University Press (1996)
- [10] Behm, P., Benoit, P., Meynadier, J.M. : METEOR : A successful application of B in a large project. In : *Integrated Formal Methods, IFM99*. Volume 1708 of LNCS., Springer Verlag (1999) 369–387
- [11] Badeau, F., Amelot, A. : Using B as a high level programming language in an industrial project : Roissy VAL. In : *ZB 2005 : Formal Specification and Development in Z and B*, 4th International Conference of B and Z Users. Volume 3455 of LNCS., Springer-Verlag (2005) 334–354
- [12] Steria : Obligations de preuve : Manuel de référence, version 3.0. Steria – Technologies de l'information. (1998)
- [13] Clearsy : B4free. website (2004) <http://www.b4free.com>.
- [14] Meyer, E., Souquières, J. : A systematic approach to transform OMT diagrams to a B specification. In : *Proceedings of the Formal Method Conference*. Number 1708 in LNCS, Springer-Verlag (1999) 875–895
- [15] Ledang, H., Souquières, J. : Modeling class operations in B : application to UML behavioral diagrams. In : *ASE'2001 : 16th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society (2001) 289–296

- [16] AFADL'2000 : Étude de cas : système de contrôle d'accès. In : Journées AFADL, Approches formelles dans l'assistance au développement de logiciels. (2000) actes LSR/IMAG.
- [17] Brown, A.W., Wallnan, K.C. : Engineering of component-based systems. In : Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), IEEE Computer Society (1996) 414
- [18] Heineman, G., Ohlenbusch, H. : An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute (February 1999)
- [19] Heisel, M., Santen, T., Souquières, J. : Toward a formal model of software components. In : Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02. Number 2495 in LNCS, Springer-Verlag (2002) 57–68
- [20] Canal, C., Murillo, J.M., Poizat, P. : Software adaptation. *L'Objet* **12**(1) (2006) 9–31
- [21] Purtilo, J.M., Atlee, J.M. : Module reuse by interface adaptation. *Software - Practice and Experience* **21**(6) (1991) 539–556
- [22] Yellin, D.D.M., Strom, R.E. : Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems* **19**(2) (1997) 292–333
- [23] Zaremski, A.M., Wing, J.M. : Specification matching of software components. *ACM Transaction on Software Engeniering Methodology* **6**(4) (1997) 333–369
- [24] Schmidt, H.W., Reussner, R.H. : Generating adapters fo concurrent component protocol synchronisation. In Crnkovic, I., Larsson, S., Stafford, J., eds. : Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems. (2002) 213–229
- [25] Reussner, R.H., Schmidt, H.W., Poernomo, I.H. : Reasoning on software architectures with contractually specified components. In Cechich, A., Piattini, M., Vallecillo, A., eds. : *Component-Based Software Quality : Methods and Techniques*. Springer-Verlag, Berlin, Germany (2003) 287–325
- [26] Poizat, P., Salaün, G., Tivoli, M. : An adaptation-based approach to incrementally build component systems. *Electronic Notes in Theoretical Computer Science* **182** (2007) 155–170
- [27] WCAT2006 : Coordination and adaptation techniques : Bridging the gap between design and implementation. In Becker, S., Canal, C., Diakov, N., Murillo, J.M., Poizat, P., Tivoli, M., eds. : *Proceedings of the Third International Workshop on Coordination and Adaptation Techniques for Software Entities*. (2006)
- [28] Mätzel, K.U., Schnorf, P. : Dynamic component adaptation. Technical report, Ubilab laboratory, Union Bank of Switzerland, Zürich, Switzerland (June 1997)
- [29] Kniesel, G. : Type-safe delegation for run-time component adaptation. *Lecture Notes in Computer Science* **1628** (1999) 351–366
- [30] Ehrig, H., Padberg, J., Braatz, B., Klein, M., Orejas, F., Perez, S., Pino, E. : A generic framework for connector architectures based on components and transformation. In : FESCA'04, satellite of ETAPS'04. Volume 108 of ENTCS. (2004) 53–67
- [31] Bracciali, A., Brogi, A., Canal, C. : A formal approach to component adaptation. *Journal of Systems and Software* **74**(1) (2005) 45–54