Assistance à l'architecte pour la construction d'architectures à base de composants

Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, Marianne Huchard

▶ To cite this version:

Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, Marianne Huchard. Assistance à l'architecte pour la construction d'architectures à base de composants. LMO 2006 - Langages et Modèles à Objets, Mar 2006, Nîmes, France. pp.37-52. lirmm-00120453

HAL Id: lirmm-00120453
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00120453

Submitted on 13 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Assistance à l'architecte pour la construction d'architectures à base de composants

Nicolas Desnos* — **Christelle Urtado*** — **Sylvain Vauttier*** **Marianne Huchard****

* *LGI2P - Ecole des Mines d'Alès - Parc scientifique G. Besse - F30 035 Nîmes*
{nicolas.desnos, christelle.urtado, sylvain.vauttier}@site-eerie.ema.fr

** *LIRMM - UMR 5506 - CNRS et Univ. Montpellier 2 - F34392 Montpellier cedex 05*
huchard@lirmm.fr

RÉSUMÉ. *Dans un processus de développement à base de composants, la sélection et l'assemblage de composants logiciels incombent à l'architecte. De nombreux travaux se focalisent sur la tâche, complexe, de vérification de la validité d'un assemblage existant. Ce papier présente un système, destiné à l'architecte, qui offre trois niveaux d'assistance imbriqués facilitant la construction d'architectures valides. Il s'appuie sur une information décrivant les collaborations potentielles entre composants, plus riche que les seules interfaces fournies et requises et plus simple et synthétique que les protocoles. Nous proposons d'enrichir la description des composants de ports, primitifs et composites, qui fournissent un niveau d'information intermédiaire à partir duquel nous pouvons vérifier une propriété des architectures, la quasi-validité, qui est une condition nécessaire à leur validité. Nous appuyant sur ces notions, nous fournissons à l'architecte une représentation abstraite des collaborations potentielles, un algorithme d'aide semi-automatisé à la construction d'architectures et un algorithme entièrement automatique.*

ABSTRACT. *During a component-based development process, the architect selects and assembles software components. Many research works focus on the complex task of verifying the validity of an existing assembly. This paper presents a computer-aided system that offers the architect three assistance levels to make the building of valid architectures easier. It is based on information about potential collaborations between components that are both richer than the usual provided and required interfaces and simpler and more synthetic than full protocols. We propose to enrich the description of components with primitive and composite ports that provide an intermediary information level that we then use to verify a quasi-validity property on architectures, which proves to be a necessary condition for their validity. On this basis, we provide the architect with an abstract representation of potential collaborations, a semi-automatic algorithm to support architecture construction and a fully automatic one.*

MOTS-CLÉS : *Développement à base de composants, construction d'architectures, assemblage, assistance à l'architecte, ports primitifs et composites, quasi-validité.*

KEYWORDS: *Component-based software, architecture construction, assembly, support to the architect, primitive and composite ports, quasi-validity*

1. Introduction

Le développement orienté composant ambitionne d'améliorer la réutilisation de composants logiciels. A ce jour, il n'existe pas de véritable consensus en ce qui concerne la définition d'un modèle de composants. Cependant, les composants sont généralement décrits par un ensemble d'interfaces qui définissent ce que le composant est capable de fournir et ce que le composant doit requérir. Les assemblages de composants sont construits en connectant leurs interfaces [OMG b, OMG a, PRO 03, BRU 04, PLá 98].
Lors du processus de développement d'une application, le concepteur produit les composants et documente (méta-données) leur comportement. Ces méta-données sont généralement décrites sous la forme de protocoles qui décrivent les séquences d'opérations valides. Un architecte construit une application en sélectionnant des composants puis en les assemblant. La cohérence d'un assemblage peut être contrôlée à deux niveaux : un niveau syntaxique, dans lequel les types des interfaces connectées sont comparés (*interface-matching*), et un niveau sémantique, dans lequel ce sont les protocoles qui sont comparés.
La plupart des travaux sur les composants mettant en œuvre ce niveau sémantique [PLá 02, HAC 04, FAR 02, PRO 03, CAR 03, BOE 02] vérifient la cohérence d'assemblages déjà existants. A notre connaissance, personne n'a étudié comment assister l'architecte durant la phase d'assemblage. Ce papier décrit un système qui fournit trois niveaux d'assistance pour aider l'architecte à construire des architectures fonctionnellement satisfaisantes et ayant de bonnes chances d'être valides.
La suite de ce papier est organisée comme suit. La section 2 étudie et compare les techniques existantes de vérification des assemblages de composants et présente la problématique abordée dans ce papier. La section 3 propose d'enrichir les composants des notions de ports primitifs et composites qui représentent une partie de l'information contenue dans les protocoles. Ces notions constituent le premier niveau d'assistance car l'information qui y est représentée permet à l'architecte de sélectionner des composants ayant de bonnes chances de former un assemblage valide. La section 4 introduit une condition, nécessaire à la validité d'un assemblage, qui est plus facile à vérifier que la compatibilité des protocoles [ALF 01]. La vérification de cette condition, intégrée au deuxième niveau d'assistance, semi-automatique, permet de proposer automatiquement des composants à l'architecte afin de compléter une architecture partielle. La section 5 décrit brièvement le troisième niveau d'assistance, automatique, qui génère automatiquement des architectures complètes. Enfin, la section 6 clôt le document en proposant une conclusion et des travaux futurs.

2. Construction d'architectures par assemblage cohérent de composants

La vérification d'un assemblage de composants vise à assurer que l'application construite à partir de parties séparées propose bien les services attendus. Ce contrôle peut être effectué à deux niveaux [PRO 03, CAR 03] : syntaxique et sémantique. Cette section étudie et compare ces aspects pour différents modèles de composants et pré-

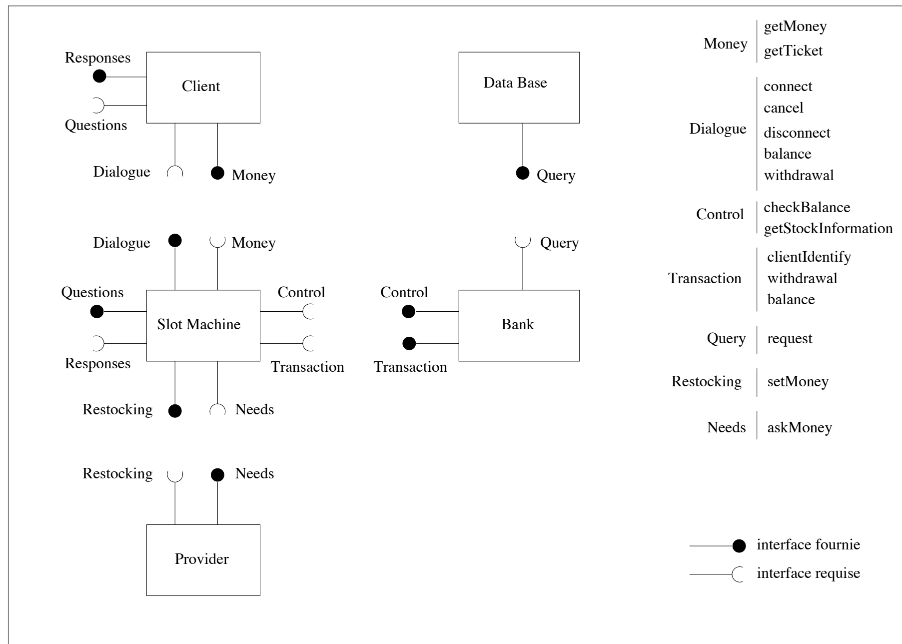

Figure 1. *Un ensemble de composants pour la construction d'un système bancaire*

sente l'objectif de ce papier : l'aide à la construction d'architectures fonctionnellement satisfaisantes et présentant de bonnes chances d'être valides. L'exemple de la figure 1 décrit un ensemble de composants destinés à être assemblés pour modéliser un système bancaire. Nous allons nous appuyer sur cet exemple dans la suite de ce document afin d'illustrer nos propos.

2.1. *Assemblage cohérent de composants*

2.1.1. *Correction syntaxique grâce aux interfaces*

Les composants sont assemblés en connectant les interfaces requises d'un premier composant aux interfaces fournies d'un deuxième composant (et vice versa). Le niveau syntaxique fournit un contrôle minimal d'un tel assemblage. Il consiste en un contrôle de compatibilité des interfaces connectées. Chaque interface est associée à un type qui correspond à un ensemble d'opérations qu'elle définit. Les types des interfaces connectées sont comparés, comme défini dans la théorie du typage utilisée dans les mécanismes de contrôle des types des langages orientés objets [DUC 02] : une interface requise doit ainsi être un super-type de l'interface fournie correspondante [OMG a, BRU 04, OPL 03].
Cependant, la correction syntaxique d'un assemblage de composants ne garantit pas

que l'assemblage peut être utilisé correctement. En effet, la correction syntaxique vérifie uniquement, sur le plan des types, que les composants seront capables d'échanger de l'information mais ne permet pas de savoir si les composants pourront collaborer, c'est-à-dire s'ils pourront échanger des informations ordonnées en séquences formant un dialogue cohérent et leur permettant de fournir les services attendus.

2.1.2. *Validité sémantique avec les protocoles*

Le niveau sémantique fournit des contrôles plus sophistiqués qui visent à garantir la robustesse du comportement dynamique d'un assemblage de composants. Ces contrôles peuvent être d'ordre non fonctionnel ou d'ordre fonctionnel [BEU 99]. Pour les aspects non fonctionnels, les propriétés que les composants doivent imposer sont généralement exprimées par des assertions. Ces propriétés peuvent résulter d'une combinaison de propriétés requises par différents composants de l'assemblage : elles sont négociées et recueillies dans des contrats qui spécifient comment les composants coopèreront réellement au travers des assemblages [BAR 03]. Notre étude se concentre sur les aspects fonctionnels du comportement d'un assemblage. Pour ce faire, la plupart des approches utilisent des protocoles qui sont des spécifications de séquences valides d'invocations d'opérations sur les composants - celles qui produisent un résultat cohérent. Les protocoles peuvent être associés à différentes sortes d'entités (interfaces, ports et composants) afin de spécifier leur comportement local ou encore à un ensemble d'entités (comme un ensemble de composants dans un assemblage ou un couple de composants collaborant) pour spécifier leur comportement global. La vérification de protocoles se ramène à des comparaisons d'automates dont la principale difficulté est l'explosion combinatoire [ALF 01].
SOFA [PLá 02] exprime les protocoles avec des expressions régulières et permet d'atteindre une définition riche et précise des protocoles, qui, en plus de permettre la description des invocations d'opérations entrantes (vers le composant) et sortantes (vers un composant connecté), permettent de rentrer dans les détails de l'exécution d'une opération interne. SOFA fournit ainsi une base formelle solide pour les différents contrôles de cohérence, basés sur l'inclusion de traces. [CAR 03] exprime les protocoles grâce à des expressions régulières et y intègre la modalité (may et must). D'autres modèles utilisent des machines à états pour spécifier les protocoles. OMEGA [BOE 02] utilise des *statecharts*. UML 2.0 [OMG b] et Java/A [HAC 04] utilisent des machines d'états de protocoles (PSM) dans le but de décrire les protocoles de façon simple. Cependant, les PSM ne peuvent prendre en compte que les appels orientés dans une même direction. Pour surmonter cette contrainte, Mencl [MEN 04] propose les PoSM, une adaptation des PSM qui permet de décrire des protocoles de ports qui sont aussi précis que ceux de SOFA. Le projet ACCORD [PRO 03] propose des contrats utilisant des machines à états. Enfin, certains modèles [WEI 01, PRO 03] utilisent aussi les assertions pour exprimer des pré et post conditions fonctionnelles qui contrôlent le séquencement des opérations.
La figure 2 montre des exemples de protocoles définissant une partie du comportement des composants de type $Client$ et $SlotMachine$. En décrivant les appels de fonctions que chacun des composants peut/doit envoyer ainsi que les appels de fonc-

Protocole du Client	Protocole de la Slot Machine
![Dialogue].connect; ou | ![Dialogue].cancel; | (![Dialogue].withdrawal; ?[Money].getMoney;)* | (![Dialogue].balance; (!?[Money].getTicket;)* ![Dialogue].disconnect;	?[Dialogue].connect{![Dialogue].clientIdentify} or | ?[Dialogue].cancel; | (?[Dialogue].withdrawal{ ![Control].checkBalance; ![Transaction].withdrawal;}; ![Money].getMoney;)* | (?[Dialogue].balance{![Transaction].balance}; ![Money].getTicket;)* or | ?[Dialogue].disconnect; | ?moneyStockTest{may[?[Restocking].setMoney{![Needs].askMoney;}; ![Control].getStockInformation;]};

LEGENDE			
	? appel reçu	may[] appel possible	or ou exclusif
	! appel émis	{ } sous protocole	?[I].y l'opération y appelée sur l'interface fournie I
	; séquence	* répétition	![I].y l'opération y partant de l'interface requise I

Figure 2. *Un exemple de protocoles*

tions qu'il accepte/exige de recevoir, les protocoles permettent de réaliser différentes vérifications sur la validité sémantique de l'assemblage de composants. Classiquement, une première propriété est la consommation des messages. Elle consiste à vérifier que toute séquence d'appels de fonction potentiellement envoyée par un composant sur une interface requise peut être exécutée par le composant qui la reçoit sur son interface fournie. D'autres propriétés plus complexes, telles que l'absence d'interblocages (cycles d'appels de fonctions entre composants) peuvent ensuite également être vérifiées [CAR 03]. Ces vérifications font appel à des procédures de génération de traces qui sont trop complexes (combinatoire importante) pour être gérées à la main et doivent donc être outillées. Même sur un exemple aussi simple que celui de la figure 1, déterminer "à la main" si le composant $Client$ et le composant $SlotMachine$ peuvent collaborer requiert une certaine expertise et un effort d'analyse des protocoles.

2.1.3. *Besoin d'un niveau de vérification intermédiaire*

La vérification syntaxique ne constitue pas une garantie suffisante de la qualité d'une architecture. La vérification sémantique s'appuyant sur des protocoles de collaboration est, quant à elle, un très bon indice de qualité mais n'est pas adaptée à une assistance à l'architecte tout au long du processus d'assemblage. En effet, la combinatoire de cette deuxième forme de vérification est acceptable (même si elle constitue un problème difficile [INV 00, ALF 01]) lorsqu'il s'agit de vérifier des connexions existantes, mais elle devient rédhibitoire lorsqu'il s'agit de choisir un composant à assembler, parmi un ensemble de composants syntaxiquement compatibles.

C'est pourquoi, nous pensons qu'il peut être intéressant de définir un niveau de contrôle intermédiaire entre le niveau syntaxique classiquement mis en œuvre grâce aux interfaces et la comparaison des protocoles.

Dans la suite de ce papier, nous dirons qu'un assemblage de composants est **syntaxiquement correct** si pour chaque connexion, les types associés aux interfaces

connectées sont compatibles et qu'il est **valide** s'il est syntaxiquement correct et si pour chaque connexion, les protocoles associés aux interfaces connectées sont compatibles.

2.2. *Architectures fonctionnellement satisfaisantes*

Une architecture est un graphe (connexe ou non) décrivant des assemblages de composants. Elle a pour objectif de rendre des services au travers des fonctionnalités de ses composants. La pertinence d'une architecture à rendre ses services est fonction du niveau d'information dont dispose l'architecte au moment de sa construction. En effet, lorsque les seules informations sur les composants sont leurs interfaces l'architecte peut :

– chercher à connecter toutes les interfaces requises par chaque composant. Dans le cas où certains composants fourniraient plus de fonctionnalités que strictement nécessaire, il est possible, étant donnée une base de composants, qu'il n'existe pas d'architecture syntaxiquement correcte. S'il en existe une, elle contiendra probablement des composants et des connexions superflues ce qui peut nuire à la qualité du logiciel résultant (baisse d'efficacité, taux d'erreur plus important), rendre l'évolution de l'architecture plus difficile (moins de potentialités que si certaines interfaces non indispensables étaient restées disponibles) et surtout, augmenter considérablement le coût de la vérification de sa validité.

– rechercher une architecture satisfaisante sans connecter toutes les interfaces, ce qui s'avère être une tâche quasiment impossible sans indication supplémentaire (connexions au hasard).

Pour palier ce problème, de nombreux modèles de composants proposent de distinguer des interfaces obligatoires et des interfaces optionnelles. Cette notion guide l'architecte dans son choix d'interfaces à connecter mais elle est associée à la nature du composant et non pas à l'usage que l'on souhaite en faire. Ainsi, elle a peu de chances d'être pertinente dans une situation donnée. La donnée de protocoles renseigne sur les différents rôles alternatifs que peut jouer un composant au sein d'une collaboration. Une architecture construite à partir de cette information ne rendra que les services souhaités et ne souffrira donc pas des défauts identifiés ci-dessus.
Les protocoles fournissent un niveau d'information suffisant pour construire de telles architectures. Dans la suite de ce papier nous désignerons par **objectif fonctionnel** l'ensemble des fonctionnalités, identifiées par l'architecte, qui doivent être supportées par une architecture et nous dirons qu'une architecture est **fonctionnellement satisfaisante** si elle est syntaxiquement correcte et si elle ne contient que des connexions nécessaires à la réalisation de son objectif fonctionnel.

2.2.1. *Besoin d'informations de granularité intermédiaire*

Les protocoles sont la seule solution existante pour construire des architectures fonctionnellement satisfaisantes. Or, nous avons conclu dans le paragraphe 2.1 que

les protocoles ne pouvaient pas être utilisés pour assister l'architecte. Il apparaît donc nécessaire d'enrichir la description des composants de concepts d'une granularité intermédiaire, plus riches que les interfaces et empruntant aux protocoles l'information sémantique sur les dépendances fonctionnelles permettant à l'architecte pour de construire des architectures fonctionnellement satisfaisantes. Cette information de granularité intermédiaire devra être le support d'un mécanisme de vérification d'une condition nécessaire à la validité d'architectures fonctionnellement satisfaisantes.
Etant donnés une bibliothèque de composants et un objectif fonctionnel, nous proposons un système fournissant à l'architecte trois niveaux imbriqués d'assistance à la construction d'architectures fonctionnellement satisfaisantes.

3. Ports primitifs et composites pour exprimer les dépendances fonctionnelles : un premier niveau d'assistance

L'objectif de cette section est de présenter le premier niveau d'assistance offert par notre système à l'architecte. Il consiste en l'ajout, dans la description des composants, de deux concepts de granularité plus appropriée que celle offerte par les interfaces pour se représenter les capacités et les besoins des composants. Ces deux concepts vont permettre d'exprimer une partie de la sémantique des protocoles.

3.1. *Dépendances entre interfaces*

Les protocoles permettent d'exprimer des séquences légales d'interactions. Il est intéressent de noter qu'un protocole décrit, de manière plus générale, des dépendances qui existent entre des interfaces. Si l'on reprend l'exemple de la figure 2, et qu'on considère le protocole du composant $SlotMachine$, la méthode $withdrawal$ appelée sur l'interface fournie $Dialogue$ appelle dans son corps la méthode $checkbalance$ par l'interface requise $Control$ suivie de la méthode $withdrawal$ par l'interface requise $Transaction$. Dans ce cas, ces trois interfaces sont dépendantes. On appelle **scénario** la description d'une séquence particulière de messages qui sont échangés par un ensemble de composants collaborant pour réaliser un objectif fonctionnel. Une condition nécessaire pour qu'un scénario soit exécutable est que toutes les interfaces des composants qui apparaissent dans ce scénario soient connectées. A l'inverse, pour qu'un scénario soit exécutable, il n'est pas la peine de connecter plus d'interfaces que celles qui figurent dans le scénario. On dit que ces interfaces sont **dépendantes**. Ce type d'information peut efficacement guider l'architecte dans son choix de connexions des composants en lui épargnant l'analyse des protocoles.
La partie gauche de la figure 3 décrit un scénario initié par le composant *Client*. Le tableau situé sur la partie droite de la figure liste les interfaces (dépendantes) qui interviennent dans l'exécution de ce scénario[1]. Dans cet exemple, les interfaces *Dialogue*

1. Dans le but de simplifier, seules les interfaces requises sont énumérées sur la figure 3. Les interfaces fournies correspondantes doivent aussi être connectées.

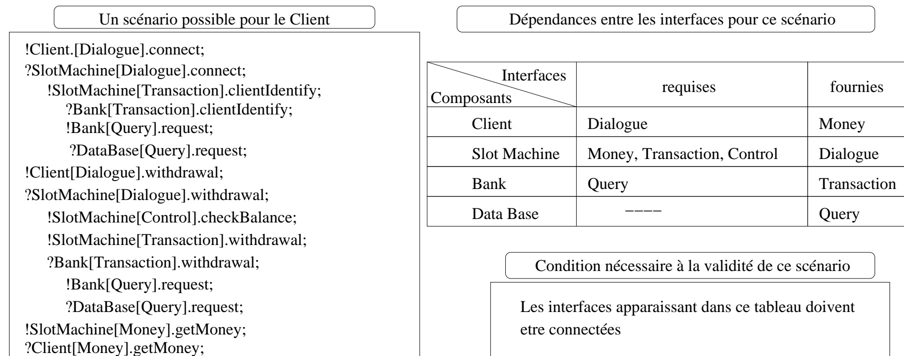

Figure 3. *Un scénario possible pour le composant Client*

et *Money* du composant *Client* doivent être connectées aux interfaces correspondantes du composant *Slot Machine* pour que le scénario soit supporté par l'architecture et les interfaces *Responses* et *Questions* du composant *Client* ne doivent pas être connectées.

3.2. *Des ports pour représenter des potentialités de collaboration*

Un port [OMG b, ALD 02, HAC 04, BOE 02, PRO 03] est une méta-information qui documente le comportement d'un composant : il décrit le rôle que peut jouer ce composant dans une collaboration. Les ports bidirectionnels regroupent les interfaces fournies et requises qui sont utilisées conjointement au sein d'une même collaboration. La connexion de deux ports réalise, en une seule opération, la connexion de toutes les interfaces requises (resp. fournies) d'un port vers les interfaces fournies (resp. requises) du port connecté. Deux ports sont syntaxiquement compatibles si toutes les interfaces (fournies et requises) de l'un trouvent dans l'autre une interface compatible. Différents modèles de composants incluent des ports. Certains [OMG b, PRO 03] autorisent l'intersection de ports – une interface peut être membre de plusieurs ports – alors que d'autres [BOE 02] non. Certains [ALD 02, HAC 04, BOE 02] imposent l'encapsulation d'interfaces – les interfaces peuvent seulement être utilisées au travers des ports dont elles font partie – d'autres [OMG b, PRO 03] non.

Les ports permettent de mettre en évidence, à un niveau structurel, une partie de l'information qui, sinon, ne figure que dans les protocoles (où elle est difficile à lire) : les dépendances entre interfaces. Cependant, la sémantique des ports telle que décrite dans les modèles de composants existants nous semble trop restrictive car elle ne permet pas de représenter des dépendances complexes comme l'illustre la partie gauche de la figure 4. Cette dernière représente les composants du système bancaire auxquels nous avons ajouté des ports primitifs. Le port primitif *Dialogue_Transaction_PrimitivePort_SM* du composant *Slot Machine* représente le rôle joué par ce composant pour réaliser un retrait. Malheureusement, il n'y a pas de port

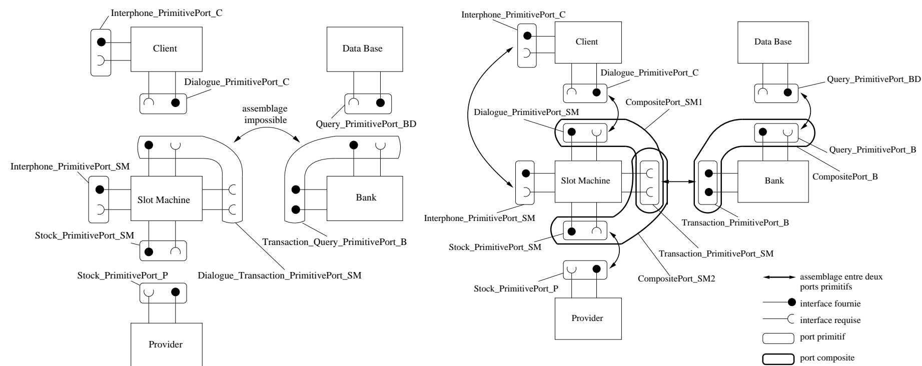

Figure 4. *Un exemple de composants : à gauche dotés de ports primitifs, à droite dotés de ports primitifs et composites*

compatible auquel le connecter. En effet, ce port correspond à deux sous-collaborations, l'une avec le composant *Client* et l'autre avec le composant *Bank*, qui ne sont pas explicitées. A l'opposé, si on choisit de décomposer le port primitif *Dialogue_Transaction-_PrimitivePort_SM* en deux ports primitifs de granularité inférieure, les dépendances entre les interfaces de ces deux ports primitifs, qui étaient représentées dans le port *Dialogue_Transaction_PrimitivePort_SM*, ne sont plus exprimées. Pour exprimer finement les dépendances fonctionnelles, il serait donc nécessaire de pouvoir disposer à la fois des ces trois ports, le port *Dialogue_Transaction_PrimitivePort_SM* étant composé des deux ports de granularité plus faible. Ce constat nous conduit à enrichir les composants d'une notion supplémentaire : celle de port composite.

3.3. *Des ports composites pour représenter des potentialités de collaborations complexes*

Les ports primitifs, composés uniquement d'interfaces, ne sont pas suffisamment expressifs dans le cas de collaborations complexes. Nous proposons d'enrichir la description des composants de ports composites pour permettre de représenter tous les types de collaborations. A notre connaissance, aucun modèle de composants ne propose de tel concept. Pour exprimer les dépendances fonctionnelles sans contraindre l'assemblage de façon excessive, la connexion de ports composites impose la connexion de ses ports composants, mais n'impose pas qu'ils soient connectés à un même composant. Sur l'exemple précédent, le port *Dialogue_Transaction_PrimitivePort_SM* est scindé en deux ports primitifs *Dialogue_PrimitivePort_SM* et *Transaction_PrimitivePort_SM* et un nouveau port composite englobant ces deux ports est ajouté. La définition que nous avons donnée à la connexion d'un port composite assure que les ports *Dialogue_PrimitivePort_SM* et *Transaction_PrimitivePort_SM* sont connectés simultanément (ce qui est nécessaire pour que le composant *Slot Machine* permette

de réaliser des retraits) mais ne sont pas contraints à être connectés à un unique composant (ce qui, nous l'avons vu ci-dessus, n'est pas possible). La partie droite de la figure 4 enrichit l'exemple initial en ajoutant des ports primitifs et composites aux composants du système bancaire et en les connectant pour former une architecture.

3.4. *Bilan*

Les notions de ports primitifs et composites telles que décrites dans cette section constituent un enrichissement des modèles de composants usuels. Nous les proposons comme concepts de granularité adaptée pour assister l'architecte lors de la sélection des composants à connecter pour construire une architecture. Plus riches que les simples interfaces, ils véhiculent de l'information sur les collaborations que les composants sont suceptibles d'établir entre eux. Plus synthétiques et lisibles que les protocoles, ils contiennent tout de même une information sur les dépendances entre interfaces, pertinente pour assister le choix des composants. Les ports primitifs et les ports composites représentent ainsi une abstraction des collaborations potentielles et des dépendances. Ils pourraient naturellement être intégrés à un outil graphique d'aide à l'assemblage de composants où ils représenteraient visuellement les informations pertinentes, habituellement enfouies dans les protocoles.

4. Quasi-validité d'une architecture : fondement du deuxième niveau d'assistance à l'assemblage

A partir des concepts proposés dans la section précédente, cette section présente le deuxième niveau d'assistance à l'architecte, lui permettant de construire de façon incrémentale (semi-automatique) des architectures fonctionnellement satisfaisantes. Etant donnés une architecture en cours de construction et un objectif fonctionnel, nous allons décrire une méthode pour vérifier si une architecture est quasi-valide. On dit d'une architecture qu'elle est **quasi-valide** pour un objectif fonctionnel donné si elle est correctement connectée, au sens des définition données ci-dessus pour la connexion de ports simples et composites. Le deuxième niveau d'assistance consiste en la vérification automatique de cette propriété pour l'architecte. Dans le cas où l'architecture proposée à la vérification par l'architecte n'est pas quasi-valide, le système informe l'architecte d'un premier ensemble de connexions à réaliser pour améliorer l'architecture. La vérification de quasi-validité s'appuie sur le parcours d'un graphe ET-OU que nous appelons graphe de voisinage.

4.1. *Graphe de voisinage d'un port primitif*

Le graphe de voisinage d'un port primitif représente l'ensemble des ports qui sont, soit directement soit indirectement, en relation avec ce port en étant soit connectés, soit dépendants (c-à-d composants d'un même port composite). Il permet de navi-

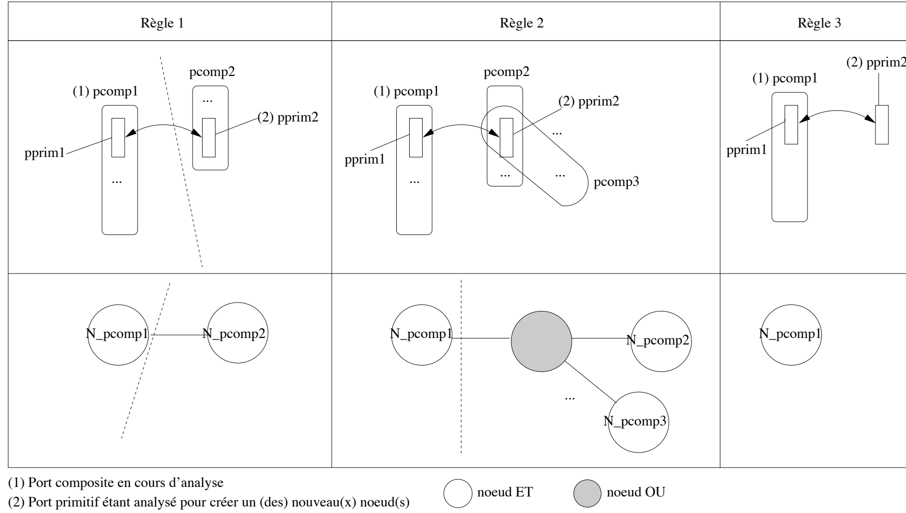

Figure 5. *Les règles de construction du graphe de voisinage*

guer dans l'architecture et de vérifier si elle est correctement connectée. Le graphe de voisinage est compact car il ne représente que les ports composites et les points de connexions alternatives, c'est-à-dire les cas où plusieurs ports composites sont candidats à la connexion. La construction du graphe de voisinage obéit aux trois règles données ci-après et illustrées sur la figure 5. Initialement, elles s'appliquent au port composite englobant le port primitif représentant l'objectif fonctionnel sélectionné par l'architecte. Ensuite, la construction s'effectue par application récursive de ces règles.

Soit N_{pcomp_1} le nœud en cours d'analyse (N_{pcomp_1} représente le port composite $pcomp_1$). Soit $pprim_1 \in pcomp_1$ le port primitif de $pcomp_1$ en cours de traitement. Soit $pprim_2$ le port primitif connecté à $pprim_1$. Initialement, N_{pcomp_1} est le port composite[2] de granularité la plus élevée englobant le port primitif exprimant l'objectif fonctionnel. Les trois règles destinées à construire le graphe de voisinage étant donnés N_{pcomp_1}, $pprim_1$ et $pprim_2$ sont :

– **Règle 1.** Si $pprim_2$ n'appartient qu'à un seul port composite, appelé $pcomp_2$, un nouveau nœud-ET appelé N_{pcomp_2} est ajouté au graphe. Une arête liant N_{pcomp_1} et N_{pcomp_2} est ajoutée.

Par exemple, cette règle s'appliquerait pour $pcomp_1$ = $CompositePort_SM2$, $pprim_1$ = $Transaction_PrimitivePort_SM$ et $pprim_2$ = $Transaction_PrimitivePort_B$.

2. Dans le cas particulier où le port primitif de départ n'est pas membre d'un port composite, nous construisons un port composite fictif l'englobant pour initier le processus.

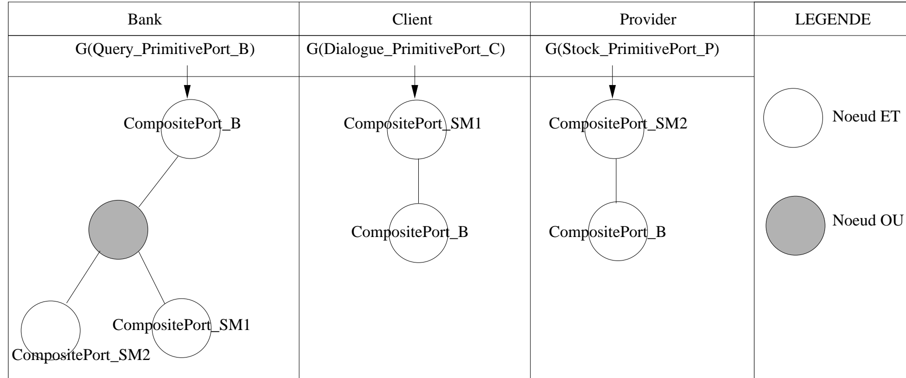

Figure 6. *Exemples de graphes de voisinage*

– **Règle 2.** Si $pprim_2$ appartient à plusieurs ports composites, un nouveau nœud-OU est ajouté au graphe. Les nœud-ET représentant tous les ports composites incluant $pprim_2$ sont ajoutés. Des arêtes sont ajoutées pour relier ces nouveaux nœuds au nœud-OU père. Pour finir, une arête reliant N_{pcomp_1} et le nœud-OU est ajoutée.
Par exemple, cette règle s'appliquerait pour $pcomp_1$ = $CompositePort_B$, $pprim_1$ = $Transaction_PrimitivePort_B$ et $pprim_2$ = $Transaction_PrimitivePort_SM$.

– **Règle 3.** Si $pprim_2$ n'appartient pas à un port composite, rien n'est ajouté au graphe.
Par exemple, cette règle s'appliquerait pour $pcomp_1$ = $CompositePort_SM2$, $pprim_1$ = $Stock_PrimitivePort_SM$ et $pprim_2$ = $Stock_PrimitivePort_P$.

La construction du graphe s'effectue en appliquant récursivement les mêmes règles à tous les ports primitifs connectés à des composants de tous les ports composites $pcomp_2$ représentés dans le graphe par un nœud N_{pcomp_2} non traité. Le graphe de voisinage obtenu constitue le support au processus de validation d'un assemblage de composants, au sens de la quasi-validité.
La figure 6 montre trois graphes de voisinages construits à partir de l'architecture présentée sur la partie droite de la figure 4. Le premier graphe de voisinage est construit à partir de l'objectif fonctionnel représenté par le port primitif $Query_PrimitivePort_B$ du composant $Bank$. Le deuxième est construit à partir du port primitif $Dialogue_PrimitivePort_C$. Le troisième est contruit à partir du port primitif $Stock_PrimitivePort_P$.

4.2. *Vérification de la quasi validité d'une architecture*

Les scénarios partant du port primitif de départ sont exécutables s'il existe au moins une transformation de son graphe de voisinage telle que décrite ci-dessous.

Intuitivement, cette transformation revient à choisir un sous graphe du graphe de voisinage pour lequel chaque nœud-OU est remplacé par une branche parmi l'ensemble des possibilités représentées par ce nœud. Le graphe résultant contient exclusivement des nœud-ET. Tous ces nœuds doivent être localement cohérents pour que le graphe soit globalement cohérent. S'il n'existe pas de tel graphe, l'architecture n'est pas quasi-valide et il est possible d'indiquer à l'architecte les premières connexions à effectuer pour tendre vers une architecture quasi-valide.

4.2.1. *Cohérence locale*

La vérification se déroule en deux temps. Tout d'abord, une vérification locale permet de déclarer un port composite donné localement cohérent. Ensuite, un parcours du graphe de voisinage permet de vérifier la cohérence globale de l'architecture à partir de la cohérence locale des ports composites et de sélections en cas de possibilités de connexions alternatives.
Un port composite est dit **localement cohérent** si tous ses ports primitifs composants (directs et indirects) sont connectés. Cette propriété est locale au port composite car elle ne considère que les connexions directes d'un port composite et ne vérifie pas l'état des ports reliés.
Soit $pprim_1$ un port primitif, nous notons $\overline{pprim_1}$ le fait que $pprim_1$ est connecté. Soit $pcomp_1$ un port composite. $pcomp_1$ est **localement cohérent** si $\forall pprim_1 \in pcomp_1, \overline{pprim_1}$.
Soit $G_p = (\mathcal{N}_{and}, \mathcal{N}_{or}, \mathcal{E}, p)$ un graphe de voisinage et $N_{pcomp_1} \in \mathcal{N}_{and}$ un nœud de G_p. N_{pcomp_1} est **localement cohérent** si le port composite correspondant $pcomp_1$ est lui-même localement cohérent.

4.2.2. *Cohérence globale*

Pour déterminer si une architecture est correctement connectée, nous procédons à une phase globale. Pour cela, le graphe de voisinage précédemment décrit est transformé de la façon suivante. Nous notons $succ_{G_p}(y)$ les successeurs du nœud y dans le graphe G_p. Soit $G_p = (\mathcal{N}_{and}, \mathcal{N}_{or}, \mathcal{E}, p)$ un graphe de voisinage. G_p est **globalement cohérent** si :

$$\left[\begin{array}{l}
\text{(1) } \exists G'_p = (\mathcal{N}'_{and}, \mathcal{N}'_{or}, \mathcal{E}', p') \text{ tel que } \mathcal{N}'_{and} = \mathcal{N}_{and}, \\
\mathcal{N}'_{or} = \oslash, p' = p, \forall (y,z) \in \mathcal{E}\text{', alors} \\
\left\{\begin{array}{l}
(a) \exists y' \in succ_{G_p}(y) \text{ tel que y' } \in \mathcal{N}_{or} \\
\text{et tel que } z \in succ_{G_p}(y') \text{ et } \forall k \in succ_{G_p}(y') \\
\text{avec } k \neq z, (y,k) \notin \mathcal{E}\text{'} \\
\text{ou} \\
\text{(b) } (y,z) \in \mathcal{E} \text{ tel que } y \in \mathcal{N}_{and} \text{ et } z \in \mathcal{N}_{and}
\end{array}\right\} \\
\text{(2) Soit } G''_p = ComposanteConnexe(G'_p, p) \\
\text{et } \mathcal{N}\text{'' l'ensemble de ses nœuds :} \\
\forall n \in \mathcal{N}'', n \text{ est localement cohérent.}
\end{array}\right]$$

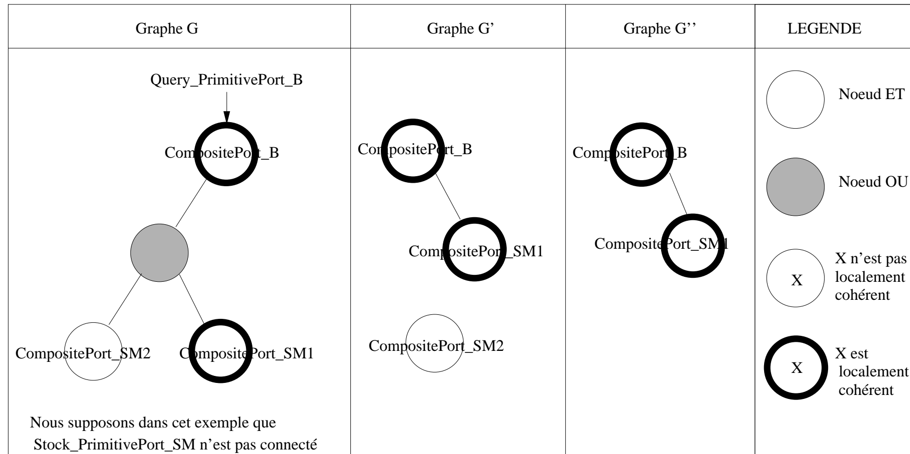

Figure 7. *Graphe de voisinage globalement cohérent*

La figure 7 illustre l'étape de vérification globale sur l'exemple précédent et montre comment G'_p et G''_p peuvent dériver d'un graphe de voisinage ${G_p}^3$.

4.2.2.1. Quasi-validité

Une architecture est quasi valide pour un objectif fonctionnel donné si le graphe de voisinage du port primitif représentant l'objectif fonctionnel est globalement cohérent. Lorsqu'il ne l'est pas, le système identifie les ports qui posent problème et recommande leur connexion à l'architecte. La propriété de quasi-validité garantit qu'une architecture a de bonne chances d'être valide (elle en est une condition nécessaire assez discriminante) et ce deuxième niveau d'assistance donne des indications interactives successives à l'architecte pour le guider lors de sa construction.

5. Troisième niveau d'assistance, implémentation et expérimentations

Le troisième niveau d'assistance procure une aide automatique à la construction d'architectures. Il consiste, à partir d'un objectif fonctionnel, à utiliser le calcul de quasi-validité pour identifier les ports qui doivent être connectés, à réaliser ces connexions et à itérer ce processus. La description détaillée de ce niveau n'est pas l'objet de ce papier. Pour réaliser nos premiers tests d'implémentation, nous avons choisi d'étendre le modèle de composants Fractal[4] [BRU 04]. Nous avons utilisé l'implémentation Ju-

3. $ComposanteConnexe(G'_p, p)$ retourne la composante connexe du graphe G'_p qui contient p.

4. Nous avons choisi Fractal car nous partageons sa vision des composants (un vrai modèle hiérarchique permettant le partage), nous apprécions sa structure (il est extensible et respecte le principe de séparation des préoccupations) et il en existe une implémentation open-source.

lia à laquelle nous avons ajouté deux contrôleurs : un contrôleur de ports primitifs dont le rôle est de créer les ports primitifs et de permettre, en une seule étape, la connexions entre ports primitifs et un contrôleur de ports composites dont le rôle est de créer les ports composites et de contrôler la quasi-validité (implémentation du deuxième niveau). Nous avons aussi réalisé un gestionnaire d'architectures qui se charge de générer automatiquement des architectures fonctionnellement satisfaisantes (implémentation du troisième niveau). Pour de petits ensembles de composants, ce gestionnaire fournit l'ensemble des architectures quasi-valides fonctionnellement satisfaisantes. Afin de faire face à la combinatoire élevée lorsque les composants et les ports sont plus nombreux, nous avons également testé plusieurs heuristiques donnant des résultats intéressants rapidement. Nous avons testés ces différents niveaux d'aide sur des composants générés en «redécoupant» des architectures générées aléatoirement.

6. Conclusion et perspectives

La contribution de ce papier consiste à fournir à l'architecte trois niveaux d'assistance à la construction d'architectures. Le premier propose les ports primitifs et composites comme concepts représentant les collaborations potentielles entre composants. Le deuxième constitue une aide semi-automatique basée sur la propriété de quasi-validité des architectures, qui est une condition nécessaire à leur validité. Le troisième niveau constitue une aide automatique qui fournit à l'architecte l'ensemble des (ou, dans le cas où la combinatoire est trop élevée, une sélection de bonnes) architectures quasi-valides fonctionnellement satisfaisantes. Nous envisageons plusieurs perspectives pour ce travail. Nous poursuivons actuellement les tests des différentes heuristiques sans lesquelles il serait inenvisageable d'intégrer notre système dans un atelier de conception. Nous pensons utiliser nos concepts comme un moyen de gérer la substitution dynamique de composants. Nous nous intéressons aussi, avec de premiers résultats intéressants, à la génération des ports à partir des protocoles.

7. Bibliographie

[ALD 02] ALDRICH J., CHAMBERS C., NOTKIN D., « ArchJava : connecting software architecture to implementation », *ICSE '02 : Proceedings of the 24th Int. Conf. on Software Engineering*, New York, NY, USA, 2002, ACM Press, p. 187–197.

[ALF 01] DE ALFARO L., HENZINGER T. A., « Interface automata », *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2001, ACM Press, p. 109–120.

[BAR 03] BARNETT M., SCHULTE W., « Runtime verification of .NET contracts », *J. Syst. Softw.*, vol. 65, n° 3, 2003, p. 199–208, Elsevier Science Inc.

[BEU 99] BEUGNARD A., JÉZÉQUEL J.-M., PLOUZEAU N., WATKINS D., « Making Components Contract Aware », *Computer*, vol. 32, n° 7, 1999, p. 38–45, IEEE Computer Society Press.

[BOE 02] DE BOER F. S., JACOB J. F., BONSANGUE M. M., « The OMEGA Component model », Deliverable of the IST-2001-33522 OMEGA project, 2002.

[BRU 04] BRUNETON E., COUPAYE T., STEFANI J., « Fractal specification - v 2.0.3 », February 2004, http ://fractal.objectweb.org/specification/index.html.

[CAR 03] CARREZ C., FANTECHI A., NAJM E., « Behavioural Contracts for a Sound Composition of Components », KÖNIG H., HEINER M., WOLISZ A., Eds., *23rd IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems*, vol. 2767 de *LNCS*, p. 111–126, Springer, Berlin, Germany, septembre 2003.

[DUC 02] DUCOURNAU R., « "Real World" as an Argument for Covariant Specialization in Programming and Modeling », *Advances in Object-Oriented Information Systems, OOIS'02 workshops*, n° 2426 LNCS, Springer, 2002, p. 3–12.

[FAR 02] FARÍAS A., SÜDHOLT M., « On Components with Explicit Protocols Satisfying a Notion of Correctness by Construction. », *CoopIS/DOA/ODBASE Proceedings*, 2002, p. 995-1012.

[HAC 04] HACKLINGER F., « Java/A - Taking Components into Java », *IASSE*, 2004, p. 163-168.

[INV 00] INVERARDI P., WOLF A. L., YANKELEVICH D., « Static checking of system behaviors using derived component assumptions », *ACM Trans. Softw. Eng. Methodol.*, vol. 9, n° 3, 2000, p. 239–272, ACM Press.

[MEN 04] MENCL V., « Specifying Component Behavior with Port State Machines. », *Electr. Notes Theor. Comput. Sci.*, vol. 101, 2004, p. 129-153.

[OMG a] OMG, « CORBA Components, Version 3.0 », http ://www.omg.org/docs/formal/02-06-65.pdf.

[OMG b] OMG, « Unified Modeling Language : Superstructure, version 2.0 », ptc/03-08-02,http ://www.omg.org/uml/.

[OPL 03] OPLUSTIL T., « Inheritance in Architecture Description Languages », *Reviewed section of Proceedings of the Week of Doctoral Students 2003 conference (WDS 2003)*, Matfyzpress, Prague, Czech Republic, June 2003, p. 124–131.

[PLá 98] PLÁSIL F., BÁLEK D., JANECEK R., « SOFA/DCUP : Architecture for Component Trading and Dynamic Updating », *CDS '98 : Proceedings of the Int. Conf. on Configurable Distributed Systems*, Washington, DC, USA, 1998, IEEE Computer Society, p. 43-52.

[PLá 02] PLÁSIL F., VISNOVSKY S., « Behavior Protocols for Software Components », *IEEE Trans. Softw. Eng.*, vol. 28, n° 11, 2002, p. 1056–1076, IEEE Press.

[PRO 03] PROJECT A., « Abstract model of contract-based component assembly », *ACCORD RNTL project number 4 deliverable*, 2003, (in french).

[WEI 01] WEIS T., BECKER C., GEIHS K., PLOUZEAU N., « A UML Meta-model for Contract Aware Components », *Proceedings of UML 2001*, London, UK, 2001, Springer, p. 442–456.