
Vers une adaptation dynamique cohérente des composants

Audrey Occello, Mireille Blay-Fornarino, Anne-Marie Dery, Michel Riveill

*ESSI/13S Université de Nice-Sophia Antipolis
930, Route des Colles- BP 145
F-06903 Sophia Antipolis Cedex
{occello, blay, pinna, riveill}@essi.fr*

RÉSUMÉ. La capacité d'une application répartie à changer de comportement en fonction de son contexte d'utilisation est essentielle. Cette évolution doit en particulier pouvoir se faire dynamiquement dans le cadre d'utilisateurs nomades. Peu de travaux, à notre connaissance, se sont intéressés à vérifier la cohérence des adaptations dynamiques, alors que différents travaux tendent aujourd'hui à la rendre opérationnelle. Lorsqu'on s'intéresse à des applications dont les composants sont interconnectés et déconnectés dynamiquement, nous constatons la nécessité d'un changement du type (au sens des messages auxquels le composant sait effectivement répondre). De par les contrôles opérés individuellement sur les composants par adaptation, nous constatons également une évolution du type des composants en terme de comportements. Il s'agit donc de s'assurer que cette évolution reste cohérente.

ABSTRACT. The capacity of distributed application to change behaviour according to the needs is essential. In particular, this evolution must be able to be done dynamically within nomad applications. Few works, to our knowledge, were interested in coherence checking of dynamic adaptations, whereas various works tend today to make it operational. When one is interested in applications whose components are inter-connected and disconnected dynamically, we note the need for a change of the type (within the meaning of the messages that the component can answer). Due to the controls operated individually on the components by adaptation, we also note an evolution of the type of the components in term of behaviours. It is thus a question of making sure that this evolution remains coherent.

MOTS-CLÉS: Typage dynamique, composants adaptables, validation et cohérence de composition
KEYWORDS: Dynamic typing, adaptable components, validation and coherence of composition

1. Introduction

La capacité d'une application répartie à changer de comportement en fonction de son contexte d'utilisation (besoins des utilisateurs, contraintes matérielles, ...) est essentielle. Cette évolution doit en particulier pouvoir se faire dynamiquement dans le cadre d'utilisateurs nomades utilisant différents types de terminaux ou des plates-formes permettant l'ajout ou le retrait d'équipement. L'adaptabilité peut porter sur différentes facettes du composant : adaptabilité à la connectivité de la plate-forme, aux ressources disponibles (réseau, mémoire, écran), à la localisation de l'utilisateur, etc. Par exemple, une application de courrier électronique s'exécutant sur un ordinateur portable doit changer dynamiquement son comportement (lecture distante ou lecture locale) selon la connectivité courante du terminal (mode déconnecté, mode connecté avec faible bande passante ou avec bande passante importante).

Ainsi, cette adaptation implique une évolution du contexte d'exécution du composant et peut être explicitée par des interactions entre le composant et son contexte. Il est alors nécessaire de prendre en charge ces interactions au niveau du composant et, par voie de conséquence, leur impact sur l'adaptation de la structure et du comportement du composant. Le comportement d'un composant dépend alors de la composition des interactions dans lesquelles il est impliqué. Or, chaque interaction pouvant imposer un type de contrôle¹ particulier et certains contrôles étant incompatibles, la composition de contrôles peut introduire des incohérences locales (ex : ajout/retrait simultanés d'une même fonctionnalité sur un composant) ou des incohérences globales par propagation des contrôles (cycles, points d'interblocage, ...).

Peu de travaux, à notre connaissance, se sont intéressés à vérifier la cohérence des adaptations dynamiques (nature de l'adaptation, moment de l'adaptation [SEG 02]), alors que différents travaux tendent aujourd'hui à la rendre opérationnelle [PAW 01, SEG 02, BER 02, BRU 00]. Si on considère qu'une application est "cohérente" lorsqu'elle réagit de façon adéquate aux messages reçus, peut-on assurer que le changement du composant par adaptation conserve cette cohérence ? Le typage dans les applications orientées Objet² n'est pas adapté à une vision évolutive du type des composants. Il permet d'assurer (au moins en théorie [DUC 02]) que les objets, de par la définition statique de leur type, sauront répondre aux messages prévus. Or, lorsqu'on s'intéresse à des applications dont les composants sont interconnectés et déconnectés dynamiquement, nous constatons la nécessité d'un changement de type (au sens des messages auxquels le composant sait effectivement répondre). De par les contrôles opérés individuellement sur les composants par adaptation, nous constatons également

1. Chaque message est intercepté lors de son émission et/ou de sa réception. Nous appelons contrôle, les exécutions qui sont faites lors de cette interception. Un contrôle peut provoquer d'autres envois de messages (notification), faire suivre l'appel à un autre objet à l'aide de la délégation, supprimer l'exécution du message (retrait), lever une exception, ajouter un nouveau message ...

2. Dans les modèles à classes, le type d'un objet de classe A peut être assimilé à l'intension de la classe [DUC 02], c'est à dire l'ensemble des propriétés de A . Ainsi, on dit qu'un objet est de type t s'il vérifie toutes les propriétés de t .

une évolution du type des composants en terme de comportements (par exemple, par la levée d'exception runtime). Il s'agit donc de s'assurer que l'évolution d'un composant reste cohérente. Le type d'un composant n'est alors plus restreint à un ensemble de messages auxquels il sait répondre, mais aussi aux "types" requis des composants avec lesquels il interagit, à des contrôles sur l'exécution des messages, à des conditionnelles sur les adaptations acceptées ou refusées . . . Dans un souci de clarté et pour éviter toute confusion avec l'approche classique du typage [CAR 96], nous désignons cette notion de "type" plus large par le terme "rôle".

L'organisation de la suite de cet article est la suivante. La section 2 présente les enjeux inhérents à une approche dynamique de l'adaptation. La section 3 décrit comment sont gérés actuellement les différents problèmes relevant de l'adaptation dynamique et propose d'aller vers une notion de type plus large pour y répondre. La section 4 expose un modèle pour composants adaptables et les propriétés qu'un tel système se doit de vérifier. Enfin, la section 5 présente nos perspectives de travail.

2. Enjeux pour une adaptation dynamique

Afin d'illustrer le propos tout au long de l'article, nous utiliserons un exemple fil rouge : une gestion d'agendas adaptables. Ainsi nous supposons disposer de composants dont le rôle est de simuler le fonctionnement d'un agenda de base c'est à dire de permettre d'ajouter, supprimer et récupérer des rendez-vous³. A l'exécution, les agendas vont être adaptés pour modifier leur comportement dans le but d'enrichir dynamiquement l'application. Notons que la construction initiale d'un assemblage de composants fait pour nous partie de l'adaptation dynamique des composants car elle peut éventuellement différer de la spécification initiale.

Ainsi, adapter dynamiquement les composants a plusieurs conséquences et permet entre autre de modifier le comportement associé aux messages. Un agenda peut être adapté pour accepter un rendez-vous uniquement sous certaines conditions (disponibilité par exemple). Cette adaptation modifie le comportement de la méthode `addRdv` de l'agenda adapté. D'autres formes d'adaptation vont faire intervenir des entités tierces, par exemple, visualiser les rendez-vous requière une IHM, les mémoriser requière une base de données. La structure du composant est modifiée en conséquence. Enfin, limiter l'accès à certains agendas pour des raisons d'authentications modifie aussi l'interface des agendas en introduisant de nouveaux paramètres (mots de passe, numéro IP) aux méthodes `addRdv` et `removeRdv`. La structure du composant est ainsi en perpétuelle évolution. Notons que même si ces extensions d'interface sont rendues transparentes vis à vis de l'utilisateur, elles restent cependant nécessaires à la gestion du composant pour les services susnommés.

L'adaptation dynamique permet donc non seulement de modifier le comportement d'un composant mais impose aussi l'évolution de son interface. Si on considère que

3. Le message `addRdv` ajoute un nouveau rendez-vous à une liste, `removeRdv` retire un rendez-vous de la liste et `getRdv` récupère la liste des rendez-vous.

le type des agendas correspond à la liste des fonctionnalités offertes, on a bien un changement du type des composants qu'il va falloir gérer.

La vérification de cette adaptation peut être locale lorsque plusieurs adaptations, qui devront être composées, portent sur la même fonctionnalité d'un composant. Par exemple, le message addRdv peut être contrôlé pour limiter l'accès à un agenda (celui de l'équipe par exemple). Il peut être adapté à nouveau pour gérer les disponibilités des membres d'une équipe en utilisant les agendas personnels de chacun. Il est par conséquent nécessaire de s'assurer que le nouveau comportement que l'on veut rajouter ne met pas le composant dans un état incohérent.

A un niveau plus global, il devient nécessaire de valider les effets d'une adaptation sur les réseaux de composants interagissant les uns avec les autres. Si un agenda est relié à une base de données et qu'ensuite l'agenda est adapté pour supporter un mode déconnecté, l'application peut se trouver dans un état incohérent si la base de données n'a pas été adaptée elle aussi pour gérer le mode déconnecté. En effet, si l'agenda pose un verrou sur la base de données et se déconnecte momentanément, certains enregistrements de la base de données se trouvent inaccessibles.

Nous allons voir dans la section suivante que ces problèmes cruciaux sont mal, voire pas du tout gérés par les approches actuelles qui font de l'adaptation.

3. Vers une notion de type plus large

Les approches permettant une adaptation des objets ou des composants par réécriture de code (adaptation statique) [KIC 01, SHI, SUN 00, MAR 02], sont basées sur une vérification des types de manière statique et n'autorisent pas l'évolution du type à l'exécution. D'autres approches basées sur le contrôle des messages à l'exécution (adaptation dynamique) [PAW 01, SEG 02], autorisent l'ajout dynamique de fonctionnalités et effectuent les vérifications de type seulement à la réception des messages⁴. Quant au retrait de message, il n'est pas pris en charge mis à part dans la gestion par exception mais ceci ne peut pas être réellement considéré comme une réduction d'interface (lever une exception c'est déjà effectuer une action).

Quant au problème de vérifier la cohérence d'une composition, il n'est que partiellement abordé parfois même, la composition n'est pas autorisée pour contourner le problème [CRU 01]. Dans les cas où le mécanisme de composition n'est pas automatisé [PAW 01, SEG 02], la charge de valider la cohérence de la composition incombe aux programmeurs, parfois, en permettant de définir, de manière programmée, des règles de composition (compatibilité, dépendance, redondance, ...) au cas par cas [PAW 01]. Quand cette tâche est automatisée, certains travaux ne prennent en charge qu'un petit nombre de situations [SUN 00, MAR 02]. Dans d'autres cas, le mécanisme de composition se base seulement sur l'ordre de déclaration des différents points

4. A terme, les langages de scripts et l'usage des invocations dynamiques dans CORBA laissent entrevoir également de telles possibilités d'évolution de type.

d'adaptation ou une relation d'ordre total entre eux pour effectuer leur composition⁵, ce qui ne garantit pas la cohérence du résultat obtenu [KIC 01]. Le plus souvent, ce problème se réduit à vérifier la cohérence de la composition localement à un composant [BER 02, PAW 01]. Les problèmes de cohérence sémantique (rafraîchissements multiples d'une même IHM) et ceux qui pourraient être engendrés à un niveau plus global tels que les cycles (boucles, provoquées par la propagation des contrôles, dans le graphe représentant les diverses interactions entre composants) ou les points de non déterminisme (ordre partiel de l'exécution des messages sur un même composant de par les interactions exercées) ne sont pas pris en compte.

Le type des composants n'est jamais étendu, même lorsque les rôles sont utilisés comme spécification supplémentaire [CRU 01], ils sont découplés du type des composants auxquels ils sont associés. Cependant élargir la notion de type pour prendre en compte des propriétés supplémentaires à vérifier nous semble être une voie intéressante pour mieux gérer l'évolution du rôle des composants et leurs compositions. L'évolution des objets vers les composants⁶ a permis d'étendre la notion d'interface pour distinguer les fonctionnalités offertes, approche usuelle des interfaces, des besoins requis, qui jusqu'ici n'apparaissaient pas, tendant ainsi à ajouter de l'information au niveau du type du composant [MAR 02, COU 02]. Par ailleurs, la programmation par contrat [MEY 92, COL 96] évoque depuis longtemps ce besoin d'élargir la notion de type à une notion plus large. En effet, les contrats étendent le type des objets en ajoutant des assertions externes qui vont compléter la spécification du type. Le principal objectif du travail amorcé, présenté dans la section suivante, consiste à rechercher une notion de "type" adéquate pour aller vers une adaptation dynamique cohérente.

Finalement, le besoin qui se fait ressentir au travers de cet état de l'art, est de disposer de plus d'information au niveau du "type" des composants pour en contrôler l'évolution et en valider la composition. Pour cela, il est nécessaire de trouver un modèle permettant de caractériser les composants et leur rôle afin de pouvoir statiquement ou dynamiquement vérifier différentes propriétés liées à l'adaptation dynamique d'un composant.

4. Modèle pour composants adaptables et propriétés à vérifier

Il existe plusieurs modèles de composants actuellement définis. Ce qui nous intéresse ici est de permettre la caractérisation des composants du point de vue de l'adaptation à l'exécution. Nous en donnons donc une définition minimale. Ainsi nous sommes amenés à définir le "type" (modèle, classe) du composant attendu en tant que rôle à remplir et le composant lui-même (instance).

5. De plus, il n'est pas toujours possible de donner un ordre, dans ce cas, il faut composer à la main les points d'adaptation non comparables.

6. "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only." [SZY 96]

Un **rôle** permet de décrire les propriétés attendues des entités auxquelles il est associé. Il se définit par un nom, par un ensemble de ports contrôlés (cibles d'une adaptation), par un ensemble de ports fournis (les fonctionnalités offertes par le composant), par un ensemble de ports émis (messages envoyés vers d'autres composants dans le cadre d'une adaptation) et par un ensemble de rôles de composants avec lesquels il interagit (via les ports émis). Un rôle peut être spécifié en forçant également d'autres propriétés liées aux comportements des entités auxquelles il est associé comme la vérification d'un automate, que ces entités soient des acteurs, qu'elles gèrent les accès concurrents (réentrance) ou au contraire un rôle pourrait être exprimé par une négation. Par exemple, pour des raisons de sécurité, un rôle peut refuser que l'entité à laquelle il doit être associé fournisse certains accesseurs.

Ainsi, le rôle d'une entité persistante est de contrôler les messages d'accès en lecture et en écriture et de faire en sorte qu'à chaque fois qu'un de ces messages est envoyé, la base de données soit notifiée et mémorise le nouvel état ou bien charge le dernier état sauvegardé. Le rôle *AgendaPersistant* se définit donc par les ports qu'il fournit (addRdv, removeRdv, getRdv), le rôle *BaseDeDonnées* qu'il requière (rôle devant fournir les ports save et load) et les ports qu'il contrôle correspondant aux fonctions d'accès en lecture/écriture (addRdv, removeRdv, getRdv).

Dans ce modèle, on distingue deux types de ports : les ports génériques et les ports instanciés. Chaque **port instancié** permet de capturer et de traiter un message particulier adressé au composant. Un **port générique** se définit par une expression désignant un ensemble de signatures de messages. En fonction des applications, ceci peut correspondre à une signature de méthode, un caractère symbolisant toute signature, une expression permettant de retrouver par exemple tous les messages dont un des arguments est d'un "type" donné. Par exemple, le port générique dont le nom commence par "get" représente l'ensemble des accesseurs. Un port générique caractérise donc un ensemble de ports instanciés.

A un **composant** est associé un rôle pouvant évoluer dans le temps (par adaptation). Tous les composants instances d'une même "classe" ont initialement le même rôle associé. Dans le cas RMI, le rôle initial se déduit des interfaces distantes implémentées par la classe de l'objet. Pour des composants EJB, le rôle initial est composé des interfaces liées à la plate-forme EJB et des interfaces métiers qu'il implémente.

Une adaptation vise à modifier le comportement d'un composant par l'introduction de contrôles sur les messages. Dans un tel modèle, une opération d'adaptation consiste à appliquer un rôle à chaque composant intervenant dans le processus d'adaptation (lorsque l'adaptation met en jeu des composants requis). Elle a pour effet de faire évoluer le rôle des composants concernés⁷. Notons que cette approche donne une autonomie plus grande au composant en lui permettant de mieux appréhender son

7. Ceci implique une évolution de structure et de comportement.

contexte d'exécution. Tout composant donné peut se prémunir d'une adaptation par l'utilisation d'assertions⁸ [COL 96].

Pour répondre aux problèmes mis en jeux lors d'une adaptation à l'exécution, voici une ébauche des propriétés qu'il faut vérifier pour que la **définition** (i.e. assemblage de composants) et l'**évolution** (i.e. conservation/restauration de l'état des composants) des composants soient cohérentes tant d'un point de vue structurel que comportemental et sémantique. Les propriétés suivantes doivent garantir qu'un envoi de message sur un composant ne puisse pas provoquer d'erreurs ("message inconnu", "incohérence de contrôles", "erreur de propagation", ...).

– Cohérence structurelle :

Tout composant requis sur un type donné doit toujours pouvoir correspondre à ce type. On utilise un composant en spécifiant le rôle que l'on attend de lui. Le rôle qu'il fournit est sous rôle⁹ du rôle attendu. Ce requis peut évoluer dans le temps pour libérer ou contraindre d'avantage le composant.

Un composant dont le rôle initial comporte les messages addRdv, removeRdv et getRdv doit toujours pouvoir répondre à ces 3 messages.

Un composant agenda que l'on veut rendre persistant doit s'assurer que le composant base de données sait répondre au message save avant de pouvoir mettre en place une connexion.

– Cohérence comportementale :

- Locale à un composant : Compatibilité des contrôles exercés.

Le comportement d'un composant doit toujours rester cohérent. Pour cela, on interdit, par exemple, de *déléguer la responsabilité de traiter un message à deux entités différentes* car suivant l'ordre des adaptations, ce ne serait pas la même délégation qui serait prise en compte.

- Locale à une adaptation : Loi du "tout ou rien".

Une opération d'adaptation est atomique. *Le rôle AgendaPersistant notifie la base de données des changements à effectuer. Si ensuite on veut gérer un mode déconnecté entre les deux, la modification des rôles associés à l'agenda et à la base de données doit se faire de manière atomique sinon des incohérences peuvent apparaître (cf. section 2).*

- Globale au réseau : Cycles, points de non-déterminisme.

Les types de contrôle susceptibles de provoquer un cycle sont les délégations et les notification. *Soit les composants a et b mis en interaction : a délègue la responsabilité de traiter le message m à b. Soit les composants b et c mis en interaction : b notifie c à chaque appel de m. Si c est ensuite adapté pour délèguer la responsabilité de traiter le message m à a, un cycle est introduit de manière indirecte par propagation des contrôles.*

– Cohérence sémantique :

Soit un composant *AgendaPersistant* requière un composant *BaseDeDonnées* fournissant la méthode "save". L'assemblage du composant *AgendaPersistant* avec un composant *Historique* qui fournit bien une méthode "save" (avec les "bons" paramètres) ne crée pas d'incohérences structurelles mais la différence de comportement d'une base de données et d'un historique peut provoquer une incohérence sémantique.

8. Par exemple, l'application du nouveau rôle doit réussir, interdiction d'appliquer certains contrôles, autoriser seulement quand le composant est dans certains états, ...

9. La relation est donnée formellement dans [OCC 02].

5. Perspectives, conclusion

Le modèle présenté dans la section précédente a été partiellement formalisé dans [OCC 02]. Actuellement, selon l'approche MDA, nous projetons ce modèle abstrait dans différents modèles de composants existants afin d'en vérifier la validité. Dans le même temps, il nous semble aussi essentiel d'apprécier l'utilité des rôles, de par leur généralité, pour faire coopérer des composants appartenant à des modèles différents et pour les réutiliser dans des contextes différents¹⁰ et d'ajouter une portée sémantique à la définition de ceux-ci.

Pour cela nous devons nous inspirer des travaux liés aux assertions, aux nomenclatures et aux ontologies pour affiner la sémantique d'utilisation des rôles et par voie de conséquence pour mieux répondre au problème de la cohérence globale. Par exemple, la représentation des graphes de composants interconnectés permettra de détecter rapidement les cycles engendrés par certaines adaptations.

Outre l'aspect immédiat sur l'évolutivité des applications adaptables, ces travaux de recherche peuvent permettre une réutilisation encore plus aisée de composants existants en permettant leur évolution en vue de leur intégration pour construire, éventuellement à plusieurs de nouvelles applications par assemblage de composants. L'utilisation des rôles peut également permettre de faciliter le travail d'intégration de projets, tâche encore difficile à réaliser aujourd'hui par trop basée sur un "matching" exact des noms de classes et de méthodes.

6. Bibliographie

- [BER 02] BERGER L., « Support des interactions dans les systèmes objets et componentiels », *Numéro spécial de L'Objet*, vol. 8, n° 3, 2002, Hermès Sciences, Coopération dans les systèmes à objets, à paraître.
- [BRU 00] BRUNETON E., RIVEILL M., « Javapod : une plate-forme à composants adaptable et extensible », rapport n° RR-3850, 2000, INRIA.
- [CAR 96] CARDELLI L., « Type systems », *ACM Computing Surveys*, vol. 28, n° 1, 1996.
- [COL 96] COLLET P., ROUSSEAU R., « Assertions Are Objects Too ! », MONNINGER F., Ed., *Proceedings of the 1st White Object-Oriented Nights Conference (WOON'96)*, St-Petersburg, Russia, Electronic proceedings (www.sigco.com/woon), 1996, p. 1–13.
- [COU 02] COUPAYE T., BRUNETON E., STEFANI J.-B., « The Fractal Composition Framework », Specification, July 2002, The ObjectWeb Consortium.
- [CRU 01] CRUZ J.-C., « CORODS : A Coordination Programming System for Open Distributed Systems », *Langages et modèles à objets LMO'2001*, vol. 7 de *L'Objet*, Le Croisic, France, January 2001, Hermès, p. 11–26.

10. Ainsi, un rôle *EntitéPersistante* peut être utilisé pour adapter et rendre persistant un agenda aussi bien qu'un compte en banque (à condition que ces derniers proposent les accesseurs et mutateurs de chaque attribut à rendre persistant).

- [DUC 02] DUCOURNAU R., « Spécialisation et sous-typage : thème et variations », *Technique et science informatiques*, , 2002, à paraître.
- [KIC 01] KICZALES G., LAMPING J., « AspectJ Home Page », <http://aspectj.org/>, 2001.
- [MAR 02] MARVIE R., PELLEGRINI M.-C., « Modèles de composants, un état de l'art », *Numéro spécial de L'Objet*, vol. 8, n° 3, 2002, Hermès Sciences, Coopération dans les systèmes à objets, à paraître.
- [MEY 92] MEYER B., « Applying "Design by Contract" », *IEEE Computer*, vol. 25, p. 40–51, 1992.
- [OCC 02] OCCELLO A., « Composants : Vers une adaptation dynamique cohérente », Rapport de DEA, juillet 2002.
- [PAW 01] PAWLAK R., SEINTURIER L., DUCHIEN L., FLORIN G., « JAC : A Flexible and Efficient Solution for Aspect-Oriented Programming in Java », A. Y., S. M., Eds., *Reflection*, vol. LNCS 2192, Springer-Verlag, 2001, p. 1-24.
- [SEG 02] SEGARRA M.-T., ANDRÉ F., « Un modèle de composants pour l'adaptation dynamique à l'environnement », *Langages et modèles à objets LMO'2002*, vol. 8 de *L'Objet*, Hermès, 2002, p. 217-229.
- [SHI] SHIBA S., « OpenC++ Home Page », <http://www.csg.is.titech.ac.jp/chiba/openc++.html>.
- [SUN 00] SUN MICROSYSTEM INC., « Enterprise Javabeans Specification. Version 1.1 », <http://java.sun.com/products/ejb/docs.html>, janvier 2000.
- [SZY 96] SZYPERSKI C., PFISTER C., « WCOP'96 Workshop Report », rapport, 1996, Workshop Reader ECOOP'96, p. 127-130.