Contributions à la vérification de la sûreté de l'assemblage et à l'adaptation de composants réutilisables

Sebti Mouelhi

UNIVERSITÉ DE FRANCHE-COMTÉ

UFR Sciences et Techniques **École doctorale SPIM**

Contributions à la vérification de la sûreté de l'assemblage et à l'adaptation de composants réutilisables

THÈSE

présentée et soutenue publiquement le 30 Août 2011

pour l'obtention du

Doctorat de l'université de Franche-Comté

(spécialité informatique)

par

Sebti Mouelhi

Composition du jury

Rapporteurs :	Franck Barbier	Professeur, Université de Pau et des Pays de l'Adour
	Frédéric Boniol	Professeur, ONERA, ENSEEIHT, INPT
Examinateurs :	Mamoun Filali Amine	Chargé de recherche CNRS, IRIT, UPS, Toulouse
	Fabrice Bouquet	Professeur, Université de Franche-Comté
Directeur de thèse :	Hassan Mountassir	Professeur, Université de Franche-Comté
Co-directeur :	Samir Chouali	Maître de conférences, Université de Franche-Comté

Laboratoire d'Informatique de l'Université de Franche-Comté — EA 4269

L I F C

Mis en page avec la classe thloria.

Remerciements

Tout d'abord, je voudrais remercier infiniment Messieurs Franck Barbier, professeur à l'Université de Pau et des Pays de l'Adour, et Frédéric Boniol, professeur à l'Université de Toulouse et l'ONERA, d'avoir rapporté la thèse. Je leur suis très reconnaissant de l'intérêt qu'ils avaient porté à mes travaux de recherche, ceci tout en ayant un regard critique, sage et constructif. Je tiens donc à leur exprimer ma profonde gratitude.

Ensuite, j'adresse des remerciements chaleureux à Messieurs Hassan Mountassir, professeur à l'Université de Franche-Comté, et Samir Chouali, maître de conférences à l'Université de Franche-Comté, mes directeurs de thèse, de m'avoir offert l'opportunité de mener à bien et diriger ma thèse au sein du Laboratoire d'Informatique de l'Université de Franche-Comté.

Je tiens à exprimer ma sincère gratitude à Hassan, de m'avoir permis par son soutien, ses nombreux conseils et sa gentillesse, de réaliser ce travail dans les meilleures conditions possibles. Je tiens à manifester, également, à Samir ma profonde reconnaissance et un immense respect. Il était à mon écoute par pure bienveillance et bon cœur en rentrant avec moi dans toutes les particularités de mon travail et en essayant toujours de me garder sur les bonnes directions. Sa présence et son aide m'ont été indispensables.

Mes remerciements vont également à Monsieur Fabrice Bouquet, professeur à l'Université de Franche-Comté d'avoir bien voulu présider le jury de thèse et examiner mon manuscrit, et à Monsieur Mamoun Filali Amine, chargé de recherche CNRS à l'IRIT et l'Université Paul Sabatier de Toulouse, d'avoir accepté de faire partie de mon jury et d'avoir apporté des remarques pertinentes à mon travail.

À cette occasion, je voudrais remercier Monsieur Ahmed Hammad mon tuteur d'enseignement durant mon service de monitorat, de m'avoir aidé à effectuer mon service d'enseignement dans les convenables circonstances. Je reconnaît aussi l'aide de tous mes collègues durant mon service en qualité d'ATER à l'IUT de Belfort-Montbéliard.

Je remercie Vincent Hugot et Benjamin Dreux de leur aide précieuse dans la conception de l'outil *ISIA*. Je remercie aussi tous mes collègues à Besançon et Montbéliard des moments agréables que nous avions partagé ensembles et de leurs qualités humaines remarquables.

Enfin, je tiens à préciser que j'avais énormément besoin de l'amour de ma chère épouse Mariem et de son support pour réaliser ce travail. Merci de sa patience et sa présence inestimables à mes côtés. Merci à ses parents et à toute ma belle famille de leurs soutien.

Merci à ma chère sœur Hajer de ses belles paroles qui diffusaient le courage dans mon âme. Merci à ma chère tante Rachida de son soutien inconditionnel et son appréhension continue jusqu'à l'achèvement de ma thèse. Merci à mes grands-parents de leurs prières permanentes ainsi qu'à toute ma famille. Merci à tous mes amis de cœur à Besançon et particulièrement Mohamed, Nadjib et Omar.

Je ne terminerai pas sans exprimer ma profonde reconnaissance à mes parents d'avoir été la cause de mon existence. Ma gratitude est immense envers mon père qui m'a incité avec ses expressions merveilleuses et ses prières d'aller toujours vers l'avant. Voilà ! je suis arrivée à mon remerciement le plus particulier : ma mère, l'être le plus cher au monde, les mots ne peuvent pas décrire tes précieuses actions et ta fatigue incroyable sans lesquelles devenir docteur restera

uniquement un rêve, cette thèse est la tienne.

Sebti Mouelhi, le 19 Septembre 2011.

À ma mère, mon père et ma chère épouse Mariem

Résumé

Cette thèse a pour objectif de proposer une approche formelle basée sur les automates d'interface pour spécifier les contrats des composants réutilisables et vérifier leur interopérabilité fonctionnelle. Cette interopérabilité se traduit par la vérification des trois niveaux : signature, sémantique, et protocole. Le formalisme des automates d'interface est basé sur une approche « optimiste » qui prend en compte les contraintes de l'environnement. Cette approche considère que deux composants sont compatibles s'il existe un environnement convenable avec lequel ils peuvent interagir correctement. Dans un premier temps, nous proposons une approche préliminaire qui intègre la sémantique des paramètres des actions dans la vérification de la compatibilité et de la substitution des composants spécifiés par des automates d'interface. Dans un second temps, nous nous somme intéressés à adapter les composants réutilisables dont les contrats sont décrits par des automates d'interface enrichis par la sémantique des actions. En ce sens, nous avons proposé un algorithme qui permet de générer automatiquement la spécification d'un adaptateur de deux composants lorsque celui-ci existe. Dans un troisième temps, nous avons augmenté le pouvoir d'expression de notre approche proposée pour vérifier l'interopérabilité et les propriétés de sûreté des composants qui communiquent par des variables définies au niveau de leurs contrats d'interface. En particulier, nous étudions la préservation des invariants par composition et par raffinement.

Mots-clés: Composants logiciels, réutilisabilité, interopérabilité, adaptation, automates d'interface, sémantique, propriétés de sûreté.

Abstract

The aim of this thesis is to propose a formal approach based on interface automata to specify the contracts of reusable components and to verify their functional interoperability. The functional interoperability is checked at three levels : signature, semantics, and protocol. Interface automata are based on an « optimistic » approach that takes into account the environment constraints. This approach considers that two components are compatible if there is a suitable environment with which they can interact properly. First, we propose an approach allowing the integration of the semantics of the action parameters in interface automata in order to strengthen the compatibility and substitution check between components. Second, we were interested in adapting reusable components whose contracts are described by interface automata enriched by the action semantics. In this context, we propose an algorithm of automatic generation of an adaptor of two mismatched components if possible. Third, we have increased the expressive power of our proposed approach to verify the interoperability and the safety properties of components that communicate by interface variables defined at the level of their contracts. In particular, we study the preservation of invariants by composition and refinement.

Keywords: Software components, reusability, interoperability, adaptation, interface automata, semantics, safety properties.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

1 Contexte du travail

Le développement de systèmes complexes dans le domaine de l'ingénierie logicielle revêt une importance capitale car il peut conduire à des processus longs et assez coûteux. La maîtrise et l'optimisation de ces processus requièrent une capacité de raisonnement importante en termes d'abstraction et de séparation des tâches. Cette discipline constitue l'une des préoccupations majeures de nos jours et notamment pour les applications logicielles à grande échelle dans le milieu industriel. L'évolution des concepts liés au développement des logiciels au cours des dernières décennies est décrite par la figure 1.

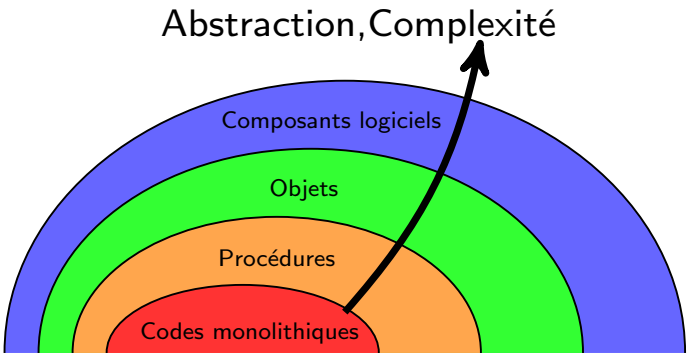


FIGURE 1 – Évolution des concepts de développement logiciel

La conception d'applications en codes monolithiques est la plus proche du langage de la machine et assure une exécution plus rapide. Par contre, le débogage de ces applications est très complexe et nécessite une connaissance approfondie des langages utilisés. Examiner un code qui contient des milliers de lignes relève de l'exploit. De plus, ces applications sont souvent non réutilisables.

L'utilisation des procédures (et fonctions) permet d'établir un niveau de modularité plus important. Les instructions d'un programme sont regroupées selon des fonctionnalités prédéfinies. La compréhension d'un programme est plus facile et, souvent, ne demande pas l'étude de l'implémentation des procédures. De ce fait, la portabilité est meilleure et le niveau de réutilisation est plus élevé. Au fil du temps, le paradigme de développement orienté objet a vu le jour et l'utilisation des procédures a évolué pour définir le comportement de tout objet ou concept de notre monde. Un objet est une unité d'instanciation qui a une identité unique et qui peut encapsuler un état et un comportement [Szy02]. Il est défini par un ensemble d'attributs, qui constituent son état variable, et un ensemble de méthodes (procédures et fonctions) qui représentent son

comportement. Un logiciel est décrit en termes de relations entre objets.

Approches à base de composants logiciels

Le développement des applications en utilisant les concepts précédemment mentionnés reste toujours une tâche très couteuse et complexe, malgré l'avantage d'être spécifiques et bien adaptées. L'idée d'acquérir ce qui est disponible pour développer ce qui est spécifique est une solution évidente pour réduire la complexité de développement des logiciels. En s'inspirant du concept des composants utilisés dans plusieurs domaines (électroniques, mécanique, ...), un logiciel peut aussi être construit par assemblage de briques logiciels réutilisables appelées *composants logiciels*. Ces dernières années, les objets ont évolué pour prendre l'identité des composants. Le composant, comme son nom le dit, c'est pour être composé et interagir avec d'autres via des interfaces fournies et requises. Un composant peut être considéré comme un programme présent en une seule instance dans un processus qui peut manipuler des objets. L'interaction entre les composants n'est pas considérée comme un simple appel de méthodes, mais plutôt comme un véritable dialogue en termes d'échange de services.

L'utilisation des composants logiciels vise à réduire d'avantage les coûts de développement d'une application par la réutilisation et l'assemblage de composants préfabriqués comme les pièces d'un puzzle. La plupart des composants ne peuvent être utilisés directement en l'état du fait qu'ils évoluent dans des environnements hétérogènes. Les techniques d'interopérabilité, d'assemblage et d'adaptation des composants apparaissent comme la solution à moindre coût pour leur réutilisation en vue de concevoir des applications logicielles de grande taille. Cependant, cette conception par composition pose souvent des problèmes de compatibilité.

Les composants logiciels sont réutilisables à travers les spécifications contractuelles de leurs interfaces. Le contrat d'un composant décrit l'essentiel suffisant de son comportement pour assurer sa communication avec ses clients (autres composants). Toutefois, les comportements des composants peuvent poser des problèmes d'incompatibilité durant leur interaction au sein d'une même architecture. Une interopérabilité sûre de deux ou plusieurs composants doit satisfaire deux propriétés :

- l'interaction des composants ne provoque pas des situations de blocage (*deadlock free*) ;
- la substitution d'un composant avec un autre plus évolué n'altère pas le fonctionnement du système.

L'ensemble des moyens nécessaires pour interconnecter les composants en synchronisant les sorties des uns avec les entrées correspondantes des autres sont appelés connecteurs. Ils peuvent être vus comme un mode de communication de deux ou plusieurs composants basée sur la synchronisation de leurs services fournis et requis. Dans le cas où l'interopérabilité implicite des composants n'est pas garantie en se basant uniquement sur leurs contrats comportementaux, une deuxième catégorie de connecteurs est envisagée. Ces connecteurs sont appelés *adaptateurs* et représentent des moyens logiciels explicites permettant d'assurer l'assemblage des composants.

De manière générale, on distingue trois niveaux d'interopérabilité fonctionnelle : signature, sémantique et protocole. Le niveau signature assure que la signature d'une opération dans le composant appelant et celle de l'opération appelée doivent être les mêmes. Le niveau sémantique concerne la compatibilité entre les sémantiques des opérations modélisées généralement par des pré et post-conditions et des invariants. Ces conditions sont des formules logiques permettant de décrire le sens et le rôle de chaque opération sans dévoiler son implémentation. Le niveau protocole concerne la vérification de l'homogénéité entre les ordonnancements temporels des

événements d'entrée et le comportement de sortie des composants.

L'utilisation de l'approche à base de composants logiciels pour le développement des systèmes complexes a exigé la nécessité de pouvoir décrire, de manière rigoureuse et efficace, les différentes étapes de spécification, de développement, de validation, et de vérification. Dans cette direction, notre conviction est la mise en avant de l'utilisation des outils mathématiques pour analyser rigoureusement la sûreté de ce genre de système. Ces outils sont appelés couramment les méthodes formelles.

Méthodes formelles

Les méthodes formelles sont présentées comme une démarche de développement basée sur des concepts de preuves mathématiques de validation et de vérification. Elles présentent l'avantage d'éviter les interprétations ambiguës dans la description d'un système, et elles garantissent la détection au plutôt de ses failles. Cette discipline a fait ses preuves pour vérifier la sûreté de la quasi totalité des systèmes informatisés, notamment les systèmes à base de composants.

Généralement, on distingue deux familles de méthodes formelles : les méthodes déductives et les méthodes basées sur des modèles formels. Dans le cadre des méthodes déductives, le système est défini par une structure mathématique et la sûreté est établie en prouvant le théorème mathématique : *spécification du système* $\Rightarrow$ *propriétés désirées*. La vérification de ce théorème est effectuée en réalisant sa preuve mathématique. Les méthodes basées sur les modèles formels (automates, systèmes de transitions, etc) décrivent le comportement du système de manière précise et explicite. Ces méthodes sont accompagnées par des algorithmes qui explorent tous les scénarios comportementaux possibles de la spécification d'un système et montrent que ses propriétés de sûreté sont satisfaites ou non.

Le contexte scientifique de cette thèse orbite autour des travaux de recherche sur la spécification et la vérification formelle des systèmes à base de composants dignes de confiance.

2 Problématiques abordées

La plupart des modèles, connus dans le monde industriel (comme les EJB,CCM,.NET), disposent d'outils permettant de vérifier l'interopérabilité des composants uniquement au niveau signatures des opérations. Le typage comportemental des interfaces au niveau sémantique et protocole reste peu abordé.

Une spécification contractuelle complète de l'interface d'un composant doit être basée sur un formalisme permettant de prendre en compte les trois niveaux de l'interopérabilité fonctionnelle. Le modèle doit décrire explicitement et de la manière la plus réaliste possible, d'une part le protocole organisant les actions d'entrée et de sortie d'un composant, et d'autre part, la synchronisation et l'entrelacement des actions des composants dans le protocole d'un système composite. De plus, le modèle doit prendre en compte les contraintes sémantiques associées aux composants durant la vérification de l'interopérabilité. Si les modèles de composants existants intègrent des spécifications d'interface basées sur des formalismes de ce genre, la vérification des propriétés de sûreté d'un système et même ses exigences non-fonctionnelles (comme les contraintes liées à la qualité de services QoS) deviennent envisageables et plus faciles à mettre en œuvre.

Plusieurs modèles ont été proposés pour la spécification des protocoles des composants logiciels notamment les automates, les machines à états, l'algèbre de processus, etc. Dans le cadre de cette thèse, nous optons pour le modèle des automates d'interface qui semble être parmi les modèles les plus naturels et expressifs pour décrire les protocoles des composants.

Formalisme des automates d'interface et ses limites

Le formalisme des automates d'interface [dAH01, AH05] permet de donner un ordre temporel sur l'enchaînement des services requis et offerts par un composant. Un appel sortant d'un composant pour solliciter un service est appelé une *action de sortie*. La réception d'un appel d'un service est appelée une *action d'entrée*. Un traitement local est appelé une *action interne*. La composition de deux automates d'interface est réalisée en synchronisant les transitions étiquetées par des actions partagées d'entrée/sortie. Une approche intéressante de vérification a également été proposée pour détecter les incompatibilités qui se produisent lorsque l'enchaînement des actions dans le produit synchronisé de deux automates conduit vers des états *illégaux* qui décrivent les situations de blocage. À partir de ces états, l'un des deux automates sollicite une action d'entrée qui n'est pas acceptée par l'autre automate. Le formalisme est soumis à une approche *optimiste* de composition. En effet, deux composants sont compatibles s'il existe un environnement qui empêche leur composition d'atteindre des situations de blocage. Deux composants sont incompatibles si tous les environnements possibles conduisent leur composition vers des situations de blocage.

Un automate d'interface spécifie le comportement légal d'un composant et en même temps il respecte les hypothèses sur le comportement de l'environnement. Pour substituer un composant par un autre plus raffiné dans un contexte, les hypothèses sur l'environnement doivent être respectées. Plus clairement, le raffinement d'un composant doit offrir plus de services et demande moins pour ne pas provoquer des situations de blocage imprévues après l'opération de substitution. Les approches de composition et de raffinement des autres formalismes à base d'automates sont considérées pessimistes car elles ne considèrent pas les contraintes de l'environnement. Il suffit de trouver un seul état de blocage pour décider de l'incompatibilité.

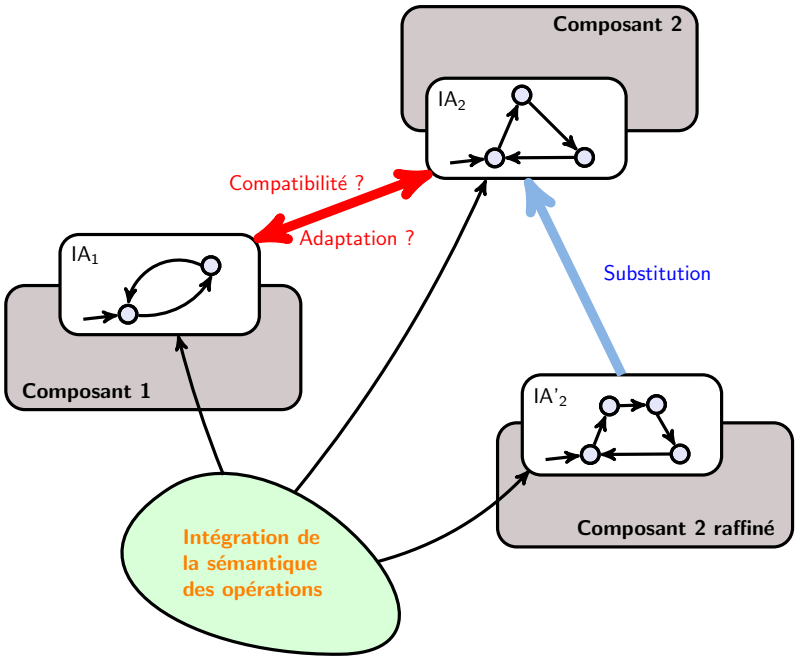


FIGURE 2 – Prise en compte de la sémantique des opérations

La vérification de la compatibilité des automates d'interface s'effectue uniquement au niveau signature sans la prise en compte des paramètres, et au niveau protocole, ce qui n'est pas suffi-

sant pour assurer une vérification fiable de l'interopérabilité. L'information sémantique doit être intégrée dans le formalisme afin d'avoir une spécification contractuelle complète des composants (cf. figure 2). Cette spécification, comme c'est déjà dit, permet de décrire de manière explicite toutes les informations nécessaires sur les composants pour les réutiliser proprement.

La sémantique d'une opération dans un composant décrit généralement l'effet de l'opération sur ses paramètres et son effet sur le comportement global du système qui encapsule le composant. En d'autres termes, une opération peut avoir deux niveaux sémantiques : le premier niveau décrit le comportement interne de l'opération (pré et post-conditions sur les paramètres) et le deuxième niveau décrit l'effet externe de l'opération après son appel sur l'évolution du comportement global du système. Ce dernier niveau peut être utilisé pour raisonner sur les propriétés de sûreté du système global.

Pour répondre à ces problématiques, nous proposons d'intégrer la sémantique des opérations dans le modèle des automates d'interface afin de fiabiliser les systèmes à base de composants.

Dans ce contexte, nous devons répondre aux questions suivantes :

- comment vérifier la compatibilité entre composants lors du processus d'assemblage du système ?
- sous quelles conditions, peut-on remplacer un composant, défaillant, par un autre plus évolué ?
- comment adapter deux composants incompatibles lors de leur réutilisation ?
- comment garantir les propriétés de sûreté des composants en se basant uniquement sur leurs interfaces contractuelles ?

3 Nos contributions

Les apports de cette thèse sont essentiellement la proposition d'approches formelles basées sur les automates d'interface pour la spécification et la vérification des systèmes à base de composants. Les approches proposées supportent les trois niveaux d'interopérabilité fonctionnelle des composants. Les contributions de la thèse sont essentiellement les suivantes :

- La première contribution consiste à proposer un formalisme basé sur l'approche optimiste des automates d'interface enrichis par la sémantique des paramètres des actions. L'approche renforce la vérification de l'interopérabilité en considérant la sémantique décrite par des pré et post-conditions des actions. La vérification de l'interopérabilité (compatibilité et substitution) se ramène à la validité d'implications entre les pré et post-conditions des actions.
- La deuxième contribution consiste à répondre à l'adaptation des composants basée sur les automates d'interface en considérant la sémantique des actions décrite en fonction des paramètres. Dans un premier temps, nous proposons une méthode qui permet de vérifier l'adaptabilité sémantique entre les actions non partagées de deux automates d'interface reliées par un contrat d'adaptation. Dans un deuxième temps, si les deux automates d'interface sont adaptables au niveau sémantique, nous générons automatiquement un adaptateur qui assure la communication entre eux. La méthode proposée combine l'approche optimiste de composition, l'intégration du niveau sémantique dans la vérification de l'interopérabilité, et les techniques d'adaptation pour le but d'établir une approche complète d'assemblage et de réutilisation des composants.
- La troisième contribution vise à augmenter le pouvoir d'expression du modèle proposé

pour pouvoir spécifier et vérifier l'assemblage des composants qui communiquent par des variables et des actions partagées. Le nouveau modèle est appelé *automates d'interface sémantiques* et décrit la sémantique des actions en fonction des paramètres et des variables. L'utilisation de ces variables permet de raisonner sur le l'évolution comportementale des composants durant le processus d'assemblage. De plus, ces variables nous permettent de vérifier les propriétés de sûreté des composants. En particulier, nous vérifions la préservation des propriétés d'invariance par la composition et par le raffinement des composants.

Pour valider notre démarche, nous présentons un environnement de spécification basé sur le langage *ISIA* (*Language for describing and checking Semantical Interface Automata*) et des techniques de vérification de propriétés de sûreté. Le langage permet de décrire la sémantique en fonction des paramètres des actions et des variables. Le cœur de l'outil permet de vérifier la compatibilité entre composants et de construire ensuite leur composition. Cette vérification est effectuée à l'aide du solveur automatique de satisfiabilité CVC3 [BT07, SBD02].

Pour vérifier les propriétés de sûreté, ces spécifications sont traduites en des modules TLA$^+$ et vérifiées à l'aide du *model-checker* TLC [Lam02, Lam94].

4 Plan du document

La thèse est structurée en trois parties. Outre l'introduction, la première partie comporte trois premiers chapitres et traite les concepts et l'état de l'art des systèmes à base de composants. La deuxième partie comporte les trois chapitres suivants et constitue les contributions de la thèse. La troisième partie est dédiée à l'outillage et à l'implantation des techniques proposées. Le document se termine par une conclusion et les perspectives de recherche à mener. Une bibliographie et des annexes sont également fournies.

Dans le chapitre 1, nous présentons en détail les termes et les concepts liés aux composants logiciels et quelques modèles de composants utilisés dans le milieu industriel. Dans le chapitre 2, nous examinons de plus prés les techniques d'assemblage et d'interopérabilité des composants. Nous explicitons les trois niveaux de l'interopérabilité fonctionnelle des composants ainsi qu'un aperçu sur les approches d'adaptation logicielle. Dans le chapitre 3, nous présentons le formalisme des automates d'interface sur lequel est basé nos travaux.

Dans le chapitre 4, nous présentons notre approche basée sur les automates d'interface enrichis par la sémantique des paramètres des actions. Le modèle présenté permet d'unifier les trois niveaux de l'interopérabilité fonctionnelle des composants. Dans le chapitre 5, nous présentons une approche d'adaptation des composants basée sur nos contributions présentées dans le chapitre 4. La technique d'adaptation proposée permet d'utiliser les informations sémantiques associées aux automates d'interface pour vérifier et valider l'assemblage des composants. Pour vérifier les propriétés de sûreté des composants, dans le chapitre 6, nous augmentons le pouvoir d'expression du modèle précédent. Ce formalisme unit l'utilisation des paramètres des opérations et les variables pour décrire les contrats des composants.

Finalement, dans le chapitre 7, nous présentons l'outil *ISIA* qui dispose d'un langage de spécification permettant de spécifier les automates d'interface sémantiques et de vérifier la compatibilité des composants et leurs propriétés de sûreté.

5 Publications

Les travaux présentés dans cette thèse ont été déjà publiés dans des conférences nationales et internationales et financés dans le cadre du projet ANR TACOS (*Trustworthy Assembling of*

Components : frOm requirements to Specification).

Les contributions présentées dans le chapitre 4 se trouvent dans [CMM10d, MCM09, CMM10c] et dans l'article de la revue *Génie Logiciel* GL [CDH+10]. Une partie des contributions présentées dans les chapitres 6 se trouvent dans [MCM11]. L'approche d'adaptation présentée dans le chapitre 5 a été publiée dans [CMM10a, CMM10b]. Une version étendue est en cours de publication dans un numéro spécial de la revue *Technique et Sciences Informatiques* TSI.

Première partie

Contexte scientifique, préliminaires

Chapitre 1

Composants logiciels

Dans ce chapitre, nous examinons de plus près les précisions techniques des termes et des concepts liés aux composants logiciels. Dans la section 1.1, nous reprenons la définition générale des composants citée dans l'introduction de la thèse et nous la raffinons en une définition plus détaillée et concrète. Dans la section 1.2, nous expliquons les notions de l'abstraction et de la réutilisation des composants. Dans les sections 1.3 et 1.4, nous évoquons les problèmes de la sémantique des interfaces des composants logiciels et nous introduisons la notion de leurs spécifications contractuelles. Dans la section 1.5, nous présentons en détails les éléments qui permettent de comprendre les protocoles comportementaux des composants. Dans les sections 1.6 et 1.7, nous présentons brièvement le concept des architectures logicielles et nous terminons le chapitre par présenter les architectures de quelques modèles à base de composants les plus connus.

1.1 Définition d'un composant logiciel

Au fil du temps, dans la littérature, les notions des classes sont adoptées par le terme « objet » et plus récemment, les objets ont commencé à prendre l'identité des « composants logiciels ». Une façon rigoureuse de définir les concepts est de dénombrer leurs caractéristiques. Dans cette section, nous allons donc définir les composants et les objets par leurs propriétés pour mieux mettre en évidence les nuances et les homologies entre leurs sens.

Plusieurs définitions ont été proposées pour définir les composants logiciels, la plus complète et celle qui a été formulée dans les proceedings [SP97] de la conférence européenne de la programmation orientée objet (ECOOP 1996). La définition en anglais est la suivante :

« *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties* ».

D'après cette définition, un composant logiciel est une unité de composition trois tiers, développé et déployable indépendamment, et destiné à être assemblé avec d'autres composants. En tant qu'une unité composable avec d'autres, un composant doit encapsuler son implémentation et interagir avec son environnement grâce à des interfaces requises et offertes bien définies. Plus précisément, il doit disposer des informations décrivant ce qu'il demande des autres composants et ce qu'il peut leur offrir comme services. Le mode d'utilisation et les contraintes, suivant lesquels les fonctionnalités d'un composant sont exécutées, représente le contrat d'un composant [Szy02]. Il est nécessaire aussi de définir ce que le contexte de composition et déploiement doit fournir

pour que les composants puissent interagir proprement. Ce contexte est composé d'un modèle de composants caractérisé par un ensemble de règles de composition et une plate-forme qui définit les règles de déploiement, de l'installation et de l'activation des composants. Les systèmes logiciels, conçus par l'assemblage d'un ensemble de composants suivant une architecture prédéfinie, sont appelés les systèmes à base de composants (*Component-based systems* CBS).

Les composants peuvent être raffinés et évolués en suivant une suite de versions. Un vendeur de composants peut proposer des différentes versions améliorées d'un composant. La gestion des versions est traditionnellement basée sur l'affectation des numéros de versions tout en gardant comme référence une seule source. L'évolution des versions est plus complexe et la gestion des numéros de versions peut devenir un problème en soi surtout au niveau de la manipulation des interfaces.

Composants et objets

Un objet est un conteneur symbolique qui incorpore des informations et des mécanismes qui symbolisent une identité physique ou morale du monde réel. La notion d'objet conduit aux notions de l'instanciation, l'encapsulation et l'identité [Szy02].

Les termes *objet* et *composant* sont souvent considérés comme synonymes, ou très similaires. Un composant peut être vu comme une collection d'objets qui communiquent les uns avec les autres. La frontière entre un composant et d'autres composants ou objets est clairement spécifiée. L'interaction avec un composant (et ainsi ses objets) à travers sa frontière est réalisée aux niveaux de ses interfaces de telle sorte que sa granularité (i.e., ses objets) reste cachée (un composant ne peut pas être utilisé pour identifier ses objets). Les objets à l'intérieur d'un composant ont accès aux implémentations des uns aux autres. Par contre, l'accès à l'implémentation d'un objet de l'extérieur du composant doit être évité [SP97].

Un composant peut contenir une ou plusieurs classes, mais une classe est nécessairement censée être une partie d'un seul composant. Un composant peut dépendre des autres composants par des relations d'importation similaires aux relations d'héritage entre les classes et les objets. La classe mère d'une classe ne doit pas nécessairement résider dans le même composant que la classe elle même. Dans le cas où la classe mère est dans un autre composant, l'héritage entre les deux classes traverse les périphéries des composants en forçant une relation d'importation entre eux [Szy02]. Au lieu de contenir des classes ou des objets, un composant peut contenir des procédures traditionnelles qui peuvent manipuler des variables globales ou il peut être développé entièrement par un langage de programmation fonctionnel.

Les propriétés suivantes représentent d'autres distinctions supplémentaires et importantes entre les composants et les objets [DW99] :

- les composants induisent l'utilisation d'un ensemble de mécanismes d'intercommunication plus étendus que les objets : les composants *Smalltalk* distribués par exemple qui supportent l'interaction de plusieurs utilisateurs distants sur plusieurs machines [CS03, Ben87, Ben88] ;
- le niveau de granularité des composants est plus large que celui des objets et leurs modes de communication via les interfaces est largement plus complexe ;
- les composants sont déployés avec différentes configurations sans avoir à les reprogrammer dans une infrastructure hébergeant.

Nous pouvons conclure que les composants sont étroitement liés aux objets et par conséquent, le développement à base de composants (*Component-Based Development* CBD) est fortement lié au développement orienté objet (*Object-oriented Design* OOD). Toutefois, de nombreux facteurs,

tels que la granularité, les concepts de composition et de déploiement et même le processus de développement, ont clairement différencié les composants des objets.

Dans la section suivante, nous détaillons les notions de l'abstraction et la réutilisation des composants qui représentent les deux caractéristiques fondamentales dans les approches de développement orienté composant CBD.

1.2 Abstraction et réutilisation des composants

L'abstraction d'un composant est le niveau de visibilité de son implémentation à travers ses interfaces. Dans une abstraction idéale d'un composant, l'environnement ne doit pas savoir aucun détail sur la spécification de ses interfaces. D'où la métaphore des "boîtes noires" (*Blackboxes*). La compréhension du fonctionnement d'un composant n'est que l'interprétation de ses sorties en fonction de ses entrées. Les tests d'un composant sont effectués selon ce principe. Les sorties délivrées, après l'exécution du composant, sont vérifiées pour décider si le composant fonctionne correctement ou non.

Cependant, les composants boîtes noires, dont les informations concernant leurs implémentations sont strictement cachées, ne sont les mieux adaptés pour une communication plus interactive avec leurs environnements logiciels. Les sorties d'un composant peuvent ne pas dépendre que de ses entrées, mais aussi des résultats d'un ensemble de services fournis par ses voisins (cf. Figure 1.1).

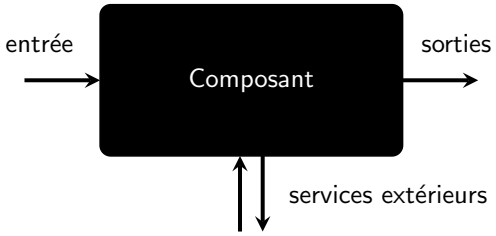


FIGURE 1.1 – Composant boîte noire

Les services extérieurs, activés par le composant, doivent être spécifiés aussi. Souvent, un tel service dépend de l'état du composant qui l'a appelé. Par exemple, le modèle de conception "observateur" [GHJV95] a besoin de demander des informations sur l'état de l'objet observé. Si l'objet observé est un compte bancaire et l'observateur est un composant de consultation de solde, on doit donc savoir si l'observateur est appelé avant ou après le changement du solde. La spécification des opérations, comme l'ajout ou la suppression de crédit, dans un composant boîte noire ne révèle pas quand est-ce que le composant observateur est appelé et l'état du composant appelant au moment de l'appel. Cet état intermédiaire du composant boîte noire peut être décrit d'une manière verbale mais dans des cas plus compliqués cette approche échoue souvent.

La question, maintenant, est comment concevoir des composants plus interactives tout en gardant cachées leurs implémentations ? La réponse est de dévoiler des parties de leurs activités intérieures (composants "boîtes grise" ou *Greyboxes*). Le composant peut donner plus de détails en cas de besoin, par exemple, l'indication des conditions sous lesquelles les services extérieurs sont appelés. Prenant l'exemple du composant observateur consultant de solde, la spécification boîte grise du composant est de clairement établir que la notification de la modification du solde est invoquée après son changement [BW97, Szy02].

En conséquence, l'abstraction reste toujours la solution qui permet à l'utilisateur d'un composant de comprendre au mieux l'essentiel de son fonctionnement plus vite. Une spécification indiquant en quelques mots que certaines données sont triées est beaucoup plus rapide à saisir que de donner les détails d'un programme de tri particulier, par exemple. De plus, les détails ignorés par le déploiement peuvent être changés plus tard sans nécessiter la modification du système de déploiement. Par exemple, il est possible de remplacer l'implémentation du tri à bulles par un autre basé sur le tri rapide.

Un composant est une unité réutilisable et il peut être utilisé dans plusieurs contextes ou être substitué par un autre qui raffine son implémentation originale en ajoutant des nouvelles fonctionnalités sans affecter ses clients (qui sont aussi des composants). Un composant peut être aussi adapté, sans la modification de son code, à un environnement particulier avec qui le composant ne peut pas communiquer directement avec ses interfaces.

1.3 Interfaces des composants

Les interfaces des composants sont les moyens par lesquelles les composants communiquent. Les descriptions des interfaces doivent être explicites de telle sorte que le rôle du composant puisse être décrit proprement. Les interfaces et leurs descriptions doivent être isolées des composants qui les implémentent [CL02, Szy02]. Cette séparation permet de (i) remplacer l'implémentation sans changer l'interface, (ii) améliorer la performance du système sans le reconstruire et (iii) ajouter des nouvelles interfaces sans changer l'existant.

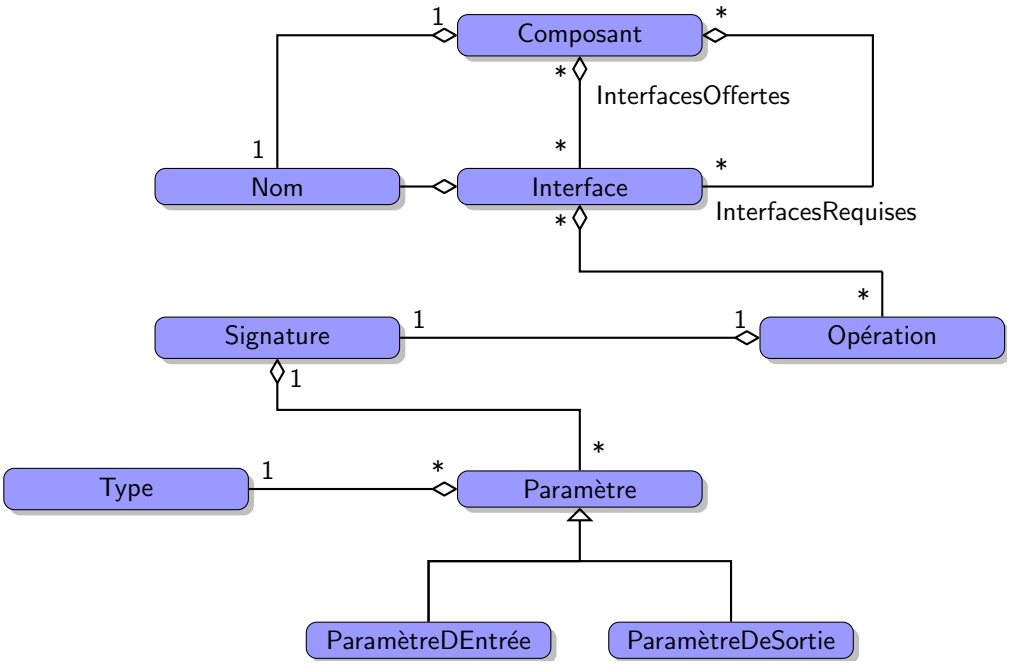


FIGURE 1.2 – Métamodèle UML des concepts de la spécification syntaxique d'un composant logiciel

Nous distinguons deux types d'interfaces : les interfaces requises et offertes. Un composant peut exporter à l'environnement des interfaces comme il peut importer d'autres de ses clients. Une interface offerte décrit un ou plusieurs service offert par le composant à ses clients, tandis que une interface requise indique un service requis par le composant de son environnement.

Concrètement, une interface est un ensemble d'opérations qui peuvent être invoquées par des clients (généralement d'autres composants). Une opération peut avoir zéro ou plusieurs paramètres d'entrée et de sortie. En général, une opération a au moins un seul paramètre de sortie (valeur de retour). Ces concepts sont illustrés par le métamodèle UML présenté dans la figure 1.2.

Un composant peut fournir une interface directement (interfaces procédurales) ou indirectement à travers des objets (interfaces d'objets). Les interfaces directement fournies par un composant sont des interfaces procédurales des librairies traditionnelles. La majorité des approches à base de composants utilisent les interfaces d'objets plutôt que les interfaces procédurales parce que le développement à base de composants est fortement lié au développement orienté objet.

À titre d'exemple, nous allons considérer des composants développés en Java. Nous n'allons pas donner les détails techniques de l'écriture des composants Java (comme par exemple les *Entreprise JavaBeans* [SM03, BMH06]) mais, nous montrons tout simplement des bouts de code de classes Java pour expliquer la notion des interfaces des composants. En général, un composant Java contient une seule classe encapsulée et porte le même nom que cette classe.

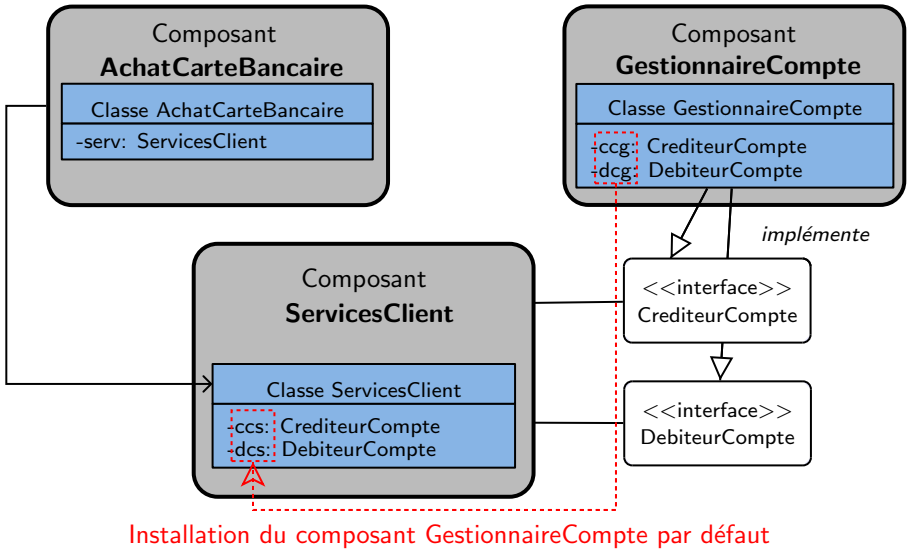


FIGURE 1.3 – Exemple des interfaces d'objets

Soit un composant qui gère les services des comptes clients d'une banque. Le composant encapsule une classe `ServicesClient` qui implémente les deux interfaces `CrediteurCompte` et `DebiteurCompte`. Le composant `ServicesClient` est aussi un médiateur pour choisir un gestionnaire de compte par défaut pour ses clients. Le composant `GestionnaireCompte` est installé par défaut en passant des références de deux objets qui implémentent les interfaces `CrediteurCompte` et `DebiteurCompte` au composant médiateur. Le gestionnaire de compte installé peut alors être utilisé indirectement par les clients du composant `ServicesClient`, par exemple un composant distant `AchatCarteBancaire` qui débite le montant d'achat d'un ensemble d'articles achetés sur Internet. La figure 1.3 illustre ce processus. Le code suivant représente la déclaration des interfaces `CrediteurCompte` et `DebiteurCompte` et la classe du composant `ServicesClient`.

```
public interface CrediteurCompte {
   /* Le composant Compte le compte d'un client en question
      dans la base de données. */
   public void crediterCompteMontant(Compte cmpt, float mnt) ;
```

```
   public void crediterCompteVirement(Compte src, Compte des, float mnt) ;
}

public interface DebiteurCompte {
   public void debiterCompteMontant(Compte cmpt, float mnt) ;
   public void debiterCompteVirement(Compte src, Compte des, float mnt) ;
}

// Entête du composant ...
public class ServicesClient implements CrediteurCompte, DebiteurCompte {
   /* Les références des objets qui implémentent
      les interfaces CrediteurCompte et DebiteurCompte. */
   private CrediteurCompte ccs ;
   private DebiteurCompte dcs ;


   public void crediterCompteMontant(Compte cmpt, float mnt){
      cmpt.crediter(mnt) ;
   }

   public void debiterCompteMontant(Compte cmpt, float mnt){
      cmpt.debiter(mnt) ;
   }
   ...
   public CrediteurCompte getCc(){
      return ccs ;
   }
   public void setCc(CrediteurCompte cc){
      this.ccs = cc ;
   }
   public DebiteurCompte getDc(){
      return dcs ;
   }
   public void setDc(DebiteurCompte dc){
      this.dcs = dc ;
   }
}
```

L'utilisation des interfaces d'objets provoque un mécanisme appelé « expédition dynamique de méthodes » (*Dynamic Method Dispatch*). Ce mécanisme permet de déterminer la bonne implémentation d'une méthode, dynamiquement durant le temps d'exécution, qui ne peut pas être déterminée d'une manière statique au moment de la compilation Ce phénomène provoque une communication tierce partie dont les clients d'un composant ignorent la présence d'un autre composant qui implémente ses interfaces. Par exemple, les clients qui utilisent les services du composant `ServicesClient` communiquent d'une façon indirecte avec le composant `GestionnaireCompte`. Le mécanisme d'expédition dynamique de méthodes induit à une communication tierce parties entre les trois composants `ServicesClient`, `GestionnaireCompte` et `AchatCarteBancaire`.

Les approches standards de description des interfaces sont en général syntaxiques et donne très peu d'informations sur leurs aspects contractuels et leurs rôles [MR98]. Toutefois, le niveau

sémantique et inter-opérationnel des interfaces reliant le composant avec son environnement (son contexte de déploiement et ses composants clients) est de nécessité fondamentale pour associer au composant une description proprement établie et complète.

1.4 Contrats

Les spécifications d'interfaces sont censées décrire ce qu'on appelle les "propriétés fonctionnelles" d'un composant. Ces propriétés sont décrites par le niveau signature des opérations fournies par le composant et le niveau comportemental par lequel le comportement du composant, en tant qu'un élément d'interaction avec d'autres, est décrit. Dans le approches orientées composant, la majorité des langages de description des interfaces, comme par exemple le langage "Interface Definition Language" (IDL) [Gud01], décrivent que le niveau signature des interfaces. Ce genre de langage ne fournit pas suffisamment d'informations sur toutes les propriétés fonctionnelles d'un composant. Par exemple, la signature de l'opération `crediterCompteVirement` de l'interface `CrediteurCompte` est définie par

```
public void crediterCompteVirement(Compte src, Compte des, float mnt) ;
```

telle que `src`, `des`, et `mnt` soient ses paramètres d'entrée dont les types sont `Compte` et `float`. La méthode n'a pas de paramètres de sortie (valeur de retour).

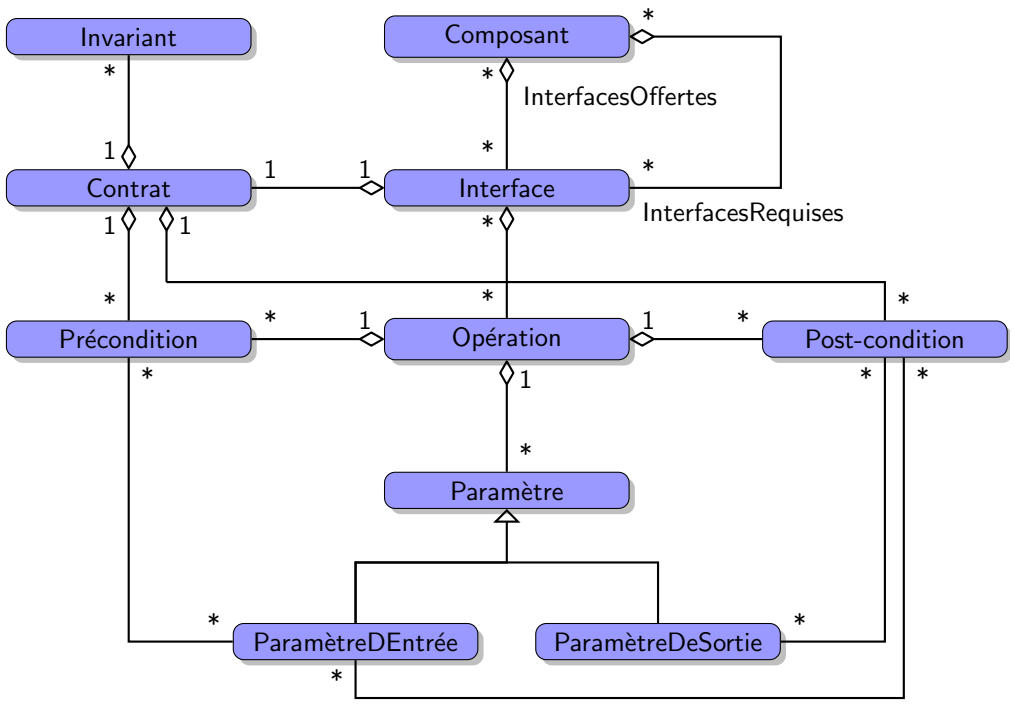


FIGURE 1.4 – Métamodèle UML des concepts de la spécification sémantique d'un composant logiciel

Les contrats sont des spécifications plus précises de l'interaction entre le client d'une interface et le fournisseur de son implémentation. Un contrat spécifie ce qu'un client doit prendre en compte pour utiliser une interface. Il indique également ce que le fournisseur doit mettre en œuvre pour satisfaire les contrainte de sortie de l'interface. Par conséquent, la spécification contractuelle d'une

interface est un modèle de convention sémantique "instancié" au moment de la composition du client et le fournisseur de l'interface. Pour chaque opération dans un composant, un contrat liste toutes les contraintes qui doivent être satisfaites par le client (les préconditions) et celles que le composant doit satisfaire en retour (les post-conditions). Selon Meyer [Mey92a], le contrat donne aussi des informations sur l'invariant qui doit être maintenu par un composant. Le métamodèle UML présenté dans la figure 1.4 illustre la spécification contractuelle sémantique des composants.

1.4.1 Sémantique des opérations

Bien que les spécifications syntaxiques des composants soient les seules les plus utilisées, il est largement reconnu que les spécifications sémantiques des opérations des composants sont nécessaires pour utiliser effectivement les composants logiciels. En effet, les technologies de composants actuelles supposent que l'utilisateur d'un composant à besoin de faire usage des informations sémantiques.

Plusieurs techniques de conception de systèmes à base de composants, qui comportent des critères sémantiques, sont présentes dans la littérature. Les travaux présentés dans [CD00] combinent les spécifications UML et le langage *Object Constraint Language* (OCL) [WK99] pour la spécification des composants. Une autre méthode bien connue qui utilise les mêmes notations est Catalysis [DW99]. Les concepts utilisés pour la spécification des composants dans Catalysis peuvent être vu comme une extension du modèle générique des spécifications syntaxiques : un composant implémente un ensemble d'interfaces, dont chacune se compose d'un ensemble d'opérations. En outre, un ensemble de pré et post-conditions sont associées à chaque opération. Ces conditions représentent communément le niveau le plus basique des spécifications sémantiques des opérations. Cette forme basique de la spécification sémantique des opérations ne donne pas d'informations sur ce qui se passe si une opération est invoquée alors que sa précondition n'est pas satisfaite. L'idée de pré et post-conditions n'est pas une nouveauté dans le développement des logiciels à base de composants. Elles sont utilisées dans une variété de techniques de développement logiciel, comme *Vienna Development Method* (VDM) [Jon90] et *Design by contract* (DbC) introduite dans [Mey88].

1.4.1.1 Pré et post-conditions

Les premiers concepts des pré et post-conditions sont attribués à Sir Tony Hoare qui, en 1969, a proposé d'ajouter aux instructions d'un programme des préconditions et des post-conditions [Hoa69]. Un *triplet de Hoare*, noté $\mathcal{P}\{\mathcal{Q}\}\mathcal{R}$, a la sémantique suivante : *Si la précondition $\mathcal{P}$ est valide avant le début du programme $\mathcal{Q}$, alors la post-condition $\mathcal{R}$ sera valide à la terminaison de $\mathcal{Q}$.* Elles étaient utilisées et mises en œuvre pour la première fois dans le langage Eiffel [Mey92b] de Bertrand Meyer.

Dans le modèle à base de composants, l'utilité des spécifications sémantiques par des pré et post-conditions est potentiellement plus large que celle des spécifications syntaxiques. Les pré et post-conditions des opérations décrivent les relations entre les paramètres d'entrée d'une opération et sa sortie. La précondition doit être vérifiée par le client avant le lancement d'une opération donnée. Cette condition doit garantir que l'exécution de l'opération est possible sans erreur. La post-condition doit être garantie par le fournisseur après le déroulement d'une opération donnée. Cette condition doit garantir que l'opération a bien réalisé son fonctionnement.

Les pré et post-conditions simples des opérations établissent seulement la correction partielle et une opération partiellement correcte peut terminer correctement ou non. Le fait qu'une opération doit également terminer, établit sa correction totale. Les conditions totalement correctes

sont formulées par les "plus faibles préconditions" $\boldsymbol{wp}(\mathcal{Q},\mathcal{R})$ (weakest preconditions) pour une opération $\mathcal{Q}$ et une post-condition $\mathcal{R}$. La précondition la plus faible est le plus petit prédicat $\mathcal{P}$ tel que $\mathcal{P}\{\mathcal{Q}\}\mathcal{R}$. Plus précisément, pour tout prédicat $\mathcal{P}$ tel que $\mathcal{P}\{\mathcal{Q}\}\mathcal{R}$, alors $\mathcal{P}$ implique $\boldsymbol{wp}(\mathcal{Q},\mathcal{R})$. La notion des plus faibles préconditions a été introduite par Dijkstra [Dij97].

Par exemple, le langage *Java Modeling Language* (JML) [LPC+08, LBR99, LC06] est utilisé pour la spécification du comportement des classes Java. La spécification décrit l'effet des méthodes sur les attributs de la classe par des pré et post-conditions définies dans le cadre de la logique de Hoare. Il décrit aussi des invariants sur le changement des états des attributs. Les spécifications sont ajoutées, en tant qu'annotations, dans les commentaires du code en Java, elles sont ensuite compilées par le compilateur Java. Les annotations JML peuvent être utilisées par exemple pour définir les contrats des composants *Enterprise JavaBeans* (EJB) des plate-formes J2EE (*Java 2 platform, enterprise edition*) [SM03, BMH06, SM96].

Prenons l'exemple de l'application qui gère les comptes clients précédemment introduits. Le composant `Compte` communique directement avec la base de données dans laquelle tous les comptes clients sont stockés. Dans le modèle des EJBs, le composant `Compte` peut être considéré comme une EntityBean qui permet de prendre en charge la persistance des données liées aux comptes clients. Les EntityBean établissent la relation entre l'application et la base de données.

Les annotations JML sont inscrites en tant que commentaires. Les pré et post-conditions des opérations sont représentées respectivement par les mots clés `requires` et `ensures`.

```
public class Compte {
   public static final float MAX_SOLDE = 100000;
   private Client c ;
   private float solde;
   private boolean isLocked = false;
   ...

   //@ requires montant > 0 && montant + solde < MAX_SOLDE;
   //@ ensures solde == \old(solde) + montant;
   public void crediter(float montant) { solde += montant; }

   //@ requires montant > 0 && montant <= solde;
   //@ ensures solde == \old(solde) - montant;
   public void debiter(float montant) { solde -= montant; }
   ...
}
```

Le langage OCL (*Object Constraint Language*) souvent associé à UML permet aussi d'exprimer les contraintes sur les opérations. Il s'agit d'un langage formel d'expression de contraintes bien adapté aux diagrammes UML, et en particulier aux diagrammes de classes. OCL peut s'appliquer sur la plupart des diagrammes UML et permet de spécifier des contraintes sur l'état d'un objet ou un ensemble objets. Les contraintes sont essentiellement des pré et post-conditions définies sur les opérations et les invariants de classes. Les pré et post-conditions du composant `Compte` peuvent être décrites de la manière suivante en OCL :

```
context Compte::crediter(montant : Real)
pre :  montant > 0 and montant + solde < MAX_SOLDE;
post : solde = solde@pre + montant;
```

```
context Compte::debiter(montant : Real)
pre :  montant > 0 and montant <= solde;
post : solde = solde@pre - montant;

context Compte::getSolde() : Real
post : result = solde
```

Les mots clés `\old` dans JML et `@pre` dans OCL, spécifie la valeur d'un attribut de classe avant l'exécution d'une méthode.

1.4.1.2 Invariants

Dans le cadre de la spécification formelle des systèmes informatiques, un prédicat est un invariant d'un programme s'il est constamment vrai durant tout le temps de son exécution. L'exécution d'un programme peut être modélisée par un ensemble d'états qui sont des valuations de variables sur lesquelles le programme agit. L'invariant est un prédicat défini sur un sous ensemble ou tout l'ensemble des variables et qui est satisfait à tous les états.

Dans le cadre des approches orientées composants, les invariants sont des conditions inscrites dans le contrat d'un composant et qui doivent être vérifiées par l'instance de ce composant durant tout son cycle de vie [Mey92a]. Le contrat donne une vue globale sur le comportement qui doit être respecté par le composant. Ces assertions sont écrites directement dans l'implémentation du composant et permettent de fournir une documentation supplémentaire sur la sémantique de ses opérations. Les propriétés d'invariance spécifient les états que les composants doivent maintenir une fois instanciés et déployés dans l'application. La satisfaction de ces propriétés est cruciale pour que l'application n'agisse pas d'une manière non désirée.

La vérification de la sûreté sur les protocoles de communication des composants est envisageable et ainsi sur l'enchaînement des événements du système composite global déduit de la composition des protocoles de ses sous-composants. Une partie de cette thèse se positionne à ce niveau. En effet, notre travail traite en particulier la vérification de la préservation des propriétés d'invariance des composants par composition et raffinement.

Prenons l'exemple précédent, un invariant possible du composant `Compte` peut être décrit de la manière suivante : respectivement en JML :

```
//@ public invariant solde >= 0 && solde <= MAX_SOLDE;
```

et en OCL :

```
context Client
inv : solde >= 0 && solde <= MAX_SOLDE.
```

1.4.2 Propriétés non-fonctionnelles

Bien que les spécifications des contrats traitent de mieux en mieux les aspects fonctionnels des composants (la syntaxe et la sémantique des interfaces, les invariants, et les pré et post-conditions), il est nécessaire de spécifier les exigences de performance et de qualité, appelées les propriétés « non-fonctionnelles » ou « extra-fonctionnelles ». Ce sont les propriétés requises pour la mise en œuvre d'un logiciel dans un environnement opérationnel. En général, les exigences fonctionnelles définissent ce qu'un système est censé faire et les exigences non-fonctionnelles définissent la manière dont le système est censé être.

La spécification contractuelle des composants doit décrire le niveau qualité de service (QoS). Ce niveau couvre les garanties concernant la disponibilité d'un composant, temps moyen entre les pannes qui peuvent survenir, temps moyen de réparation, le débit, la latence, la sécurité des données, la capacité, la consommation d'énergie (pour les systèmes à base de composants embarqués), etc. On peut s'attendre à ce que les spécifications des propriétés non-fonctionnelles dans les contrats des composants seront rigoureusement mieux traitées dans l'avenir et deviendront plus largement utilisées [GS05].

Les propriétés non-fonctionnelles de la qualité de service sont répertoriées en deux catégories principales :

1. qualités d'exécution (timing), telles que la sécurité et la facilité d'utilisation, qui sont observables au moment de l'exécution.
2. testabilité, maintenance, l'extensibilité, qui sont incorporées dans la structure statique du système logiciel [Wie03, You00].

Plusieurs travaux de recherche ont été effectués dans le cadre de la spécification des propriétés non-fonctionnelles des systèmes architecturaux à base de composants. Dans le travail présenté dans [BM03], les auteurs mettent en évidence une distinction claire entre les propriétés fonctionnelles et les propriétés non-fonctionnelles comme la fiabilité et la qualité de service (QoS) des contrats des composants. Leur travail déduit que l'adaptabilité des composants, comme étant une propriété fondamentale pour assurer la réutilisabilité, peut être vue comme étant une propriété de performance. Ils proposent une architecture à base de composants modélisée en RT-UML *Profil* pour valider les exigences de performance. Le profile RT-UML est destiné pour fournir une structure unifiée qui englobe les méthodes d'analyse existantes en laissant la flexibilité de développement par divers langages. Le travail proposé donne une argumentation claire de l'influence d'ajouter les critères liés aux propriétés non-fonctionnelles dans l'architecture abstraite d'un système.

Dans le travail proposé dans [EDG+06], le groupe OMG a proposé un nouveau profil UML, nommé MARTE (*Modeling and Analysis of Real-Time and Embedded systems*) et issu des deux profiles existants SPT et QoS&FT, pour représenter les propriétés non-fonctionnelles. Le modèle proposé supporte la vérification temporelle des modèles UML. Les auteurs discutent les annotations de quelques exigences non-fonctionnelles et comment le profil UML proposé est utilisé pour modéliser des propriétés comme la qualité de service, le temps, la tolérance aux pannes, etc.

Dans [AHS02], les auteurs présentent la théorie des interfaces temporisées (*Timed Interfaces*) qui permettent à la fois de spécifier les protocoles des composants et les contraintes liées au temps d'exécution de leurs actions. Une interface temporisée spécifie le temps requis par les entrées (services fournis ou réceptions des messages) d'un composant et celui requis par ses sorties (services requis ou envoi de messages). Deux interfaces de composants sont compatibles s'il existe un moyen de les combiner ensemble de telle sorte que le *timing* de leurs événements d'entrée et de sortie partagées soit respecté.

1.5 Protocoles comportementaux

Dans cette section, nous présentons les concepts concernant les protocoles des composants.

1.5.1 Définition et rôles

Un « protocole comportemental » définit le mode d'emploi du composant. En effet, il définit un ordre temporel sur l'enchaînement des appels des services requis et offerts par le composant.

Les protocoles comportementaux ont été introduits pour la première fois par Daniel Yellin et Robert E. Storm dans [YS97].

Le protocole décrit les interactions entre un composant et ses clients. Les interactions sont principalement des demandes pour exécuter des services offerts par l'environnement du composant ou des événements d'acceptation des requêtes pour solliciter des services offerts par le composant lui même. Il a pour but principal d'éviter l'utilisation hasardeuse des services des composants. En explicitant cette dynamique de communication avec un composant, on réduit la combinatoire d'utilisation de ses services. En particulier, les préconditions des services permettent d'abstraire un protocole comportemental implicite. Autrement dit, le protocole le plus générique consiste à utiliser n'importe quel service dans n'importe quel ordre. Les éventuelles conditions d'application se trouvent alors dans les préconditions ou les gardes des opérations. Le fait qu'une précondition d'une opération soit satisfaite à un moment donné veut dire qu'elle puisse ne pas être satisfaite après ou avant l'exécution d'une autre opération. Le métamodèle UML présenté dans la figure 1.5 illustre les concepts élémentaires utilisés pour la spécification des protocoles des composants. Si l'opération est inclue dans une interface requise, alors elle est associée à une action de sortie. Sinon, elle est associée à une action d'entrée.

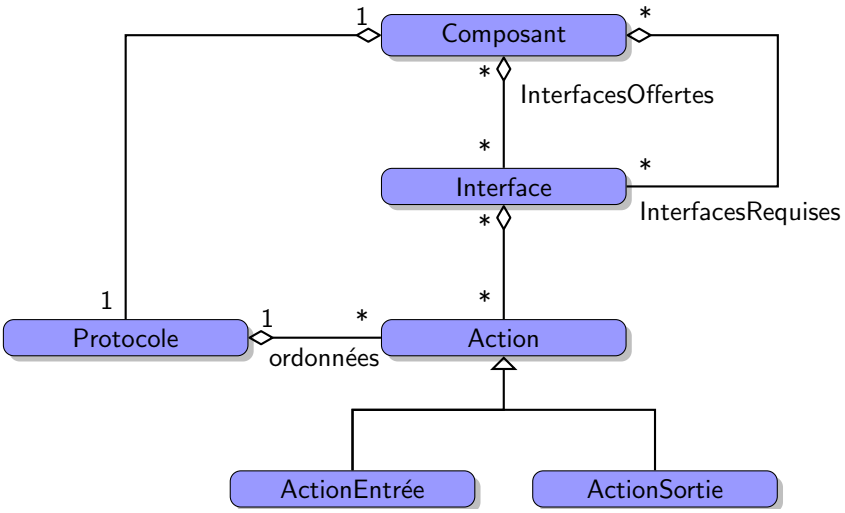


FIGURE 1.5 – Métamodèle UML des concepts de la spécification élémentaire d'un protocole de composant

Un autre objectif des protocoles est de définir un niveau de granularité plus large que celui des services et de favoriser ainsi l'abstraction, la réutilisabilité et la modularité des composants. Les protocoles des composants facilitent aussi la vérification des propriétés de cohérence des composants et de conformité de l'assemblage de composants. En effet, il ne suffit pas d'appeler correctement (compatibilité des pré et post-conditions) les services pour être sûrs que l'assemblage fonctionne, on doit aussi vérifier si l'évolution dynamique des échanges entre composants ne conduit pas à des états de blocage ou des situations indésirables [AAA07].

Par exemple, pour modifier une ressource partagée par un client, il faut d'abord envoyer une requête pour avoir la ressource, puis la modifier. La modification doit se faire en exclusion mutuelle. Dans un distributeur de billets, pour réaliser des opérations de retrait, de consultation de solde, de dépôt, ou de transfert, l'usager doit d'abord s'identifier et effectuer après les opérations qu'il souhaite faire. Ce protocole déroule dans des sessions.

Nous tenons à préciser que le travail présenté dans cette thèse se focalise sur à l'exploitation des niveaux sémantique et protocole pour la vérification de la compatibilité des composants.

1.5.2 Exemple de langages de description de protocoles

La notion de protocole est régulièrement associée à la notion du comportement dynamique d'un composant comme nous venons de détailler au début de la section. Cependant, elle varie d'un modèle de description à un autre et couvre des réalités différentes (processus de composant, protocole de communication entre deux ou plusieurs composants, contexte d'adaptation de connecteurs, etc). Pour clarifier le concept, nous proposons une brève comparaison entre les approches selon les quatre critères suivants :

- le contenu des protocoles : invocations de services, envois de messages, etc ;
- les unités rattachées aux protocoles : le composant, l'interface, le service, le connecteur ou l'architecture ;
- les formalismes de description : automates, machines à états, algèbre de processus, expressions régulières, etc.

Dans les sous-sections suivantes, nous allons présenter brièvement une étude bibliographique sur quelques formalismes communément utilisés pour la spécification des protocoles de composants. D'autres approches sont présentées progressivement tout au long des deux chapitres suivants.

1.5.2.1 CSP

Dans [AG97], les auteurs proposent une base formelle pour la spécification des interactions entre les composants en définissant une théorie et des notations qui associent, aux connexions architecturales entre les composants, une sémantique explicite. Plus précisément, le modèle intègre un système formel pour la spécification des types de connecteurs d'architecture (cf. section 2.2). La description de ces types de connecteurs est utilisée pour caractériser les protocoles d'interaction entre les composants dans une architecture logicielle. Les fondements formels pour la définition des protocoles sont basés sur CSP [Hoa78]. Le modèle montre comment les formalismes qui sont traditionnellement utilisés pour caractériser la communication par échange de messages dans le réseau, peuvent être appliqués pour la description des interactions entre composants.

Le modèle suivant décrit un système Client/serveur simple écrit en Wright (langage de description d'architecture développé par Allen Robert J. dans [AG97])

System SystemeClientServeurSimple
  component Serveur =
    port service [*protocole*]
    spec [*Spécification du serveur*]
  component Client =
    port requete [*protocol*]
    spec [*Spécification du client Serveur*]
  connector C-S-connecteur =
    role client [protocole du client]
    role serveur [protocole du serveur]
    glue [protocole de connexion]
Instances

s : Serveur
c : Client
cs : C-S-connecteur

Attachments

s.service **as** cs.serveur ;
c.requete **as** cs.client

end SystemeClientServeurSimple

La première partie de la description définit le composant et le type de ses connecteurs. Un composant est décrit par un ensemble de ports et une spécification qui modélise le comportement abstrait du composant. Chaque port décrit un point logique de communication entre le composant et son environnement. Dans l'exemple, le client et le serveur ont chacun un seul port de communication, mais en général un composant peut avoir plusieurs ports. Ainsi, les ports représentent les interfaces du composant.

Un connecteur est définit par un ensemble de rôles et une spécification « *glue* » de collage. Les rôles décrivent le comportement local prévu de chaque partie communicante. Ils déterminent les obligations de chaque composant participant à une communication.

Par exemple, le connecteur client-serveur (C-S-connecteur) a un rôle client et un rôle serveur. Le rôle du client décrit le comportement du client comme une séquence d'une alternance de demandes et de réceptions de service. Le rôle du serveur décrit le comportement du serveur comme étant une alternance entre le traitement des demandes des envois des services demandés. Le *glue* adapte le rôle du client avec celui du serveur. Cela voudrait dire que les activités doivent être planifiées dans un certain ordre : le client demande un service, le serveur gère la demande, le serveur fournit le service, et le client obtient le service.

La deuxième partie de la spécification représente un ensemble d'instance de composant et de connecteur. Elle décrit les entités effectives qui vont apparaître dans la configuration. Dans l'exemple, il y a une seule instance du composant Serveur (s), une seule instance du composant Client (c), et un seul connecteur (cs). La troisième partie de la spécification présente comment les instances de composants et le connecteur sont combinées en prescrivant les attachements entre les ports et les rôles des connecteurs. Le connecteur cs coordonne le comportement des ports c.requete et s.service.

Les auteurs utilisent une variante de CSP pour décrire les protocoles associés aux rôles des connecteurs. Nous n'allons pas donner tous les détails sur le formalisme mais tout simplement fournir les spécifications des rôles et les expliquer. Le rôle du Client est décrit par la succession de l'envoi d'une requête et la réception d'un service. Le rôle du Serveur est la réception d'une requête du client et l'envoi du service requis à plusieurs reprises. L'opérateur externe de choix ($\Box$) permet de choisir entre l'engagement dans un nouveau invoke et la terminaison provoquée par l'environnement. En d'autres termes, le serveur n'est pas autorisé à suspendre son exécution à moins que l'environnement, dans lequel il opère, permette également la terminaison.

connector C-S-connecteur =
role Client = (request !x $\rightarrow$ result ?y $\rightarrow$ Client) $\sqcap$
role Serveur = (invoke ?x $\rightarrow$ return !y $\rightarrow$ Serveur) $\Box$
glue = (Client.request ?x $\rightarrow$ Serveur.invoke !x $\rightarrow$ Serveur.return ?y $\rightarrow$ Client.result !y $\rightarrow$ **glue**)$\Box$

Le rôle du Client décrit son comportement lorsqu'il demande un service du Serveur. D'une manière similaire, le rôle du Client est un processus qui peut invoquer un service et le recevoir ensuite plusieurs fois, ou terminer. Cependant, à cause de l'utilisation de l'opérateur $\sqcap$, le choix

de savoir s'il faut appeler un service ou mettre fin à l'exécution est déterminée par le processus Client. Le processus glue adapte les deux rôles du Client et du Serveur.

1.5.2.2 π-calcul

Le travail proposé par Carlos Canal et al. [CFTV00] énonce une approche qui étend les descriptions CORBA IDL des interfaces de composants par la spécification des protocoles en utilisant l'algèbre de processus π-calcul [Mil99]. La sémantique de l'algèbre de processus est bien adaptée pour la spécification de l'évolution des systèmes communicants. Le π-calcul permet de spécifier l'échange de messages ou l'invocation des opérations par des noms et les protocoles par des processus qui représentent le cycle de vie d'un composant.

Nous commençons par présenter la variante de π-calcul décrite dans l'article. Si `ch` est un nom de canal, alors `ch!(v).P` représente un processus qui envoie une valeur `v` le long de `ch` et après il revient à son état d'origine. Contrairement, `ch?(x).Q` est le processus qui attend la réception des valeurs `v` du canal de communication `ch`, affecte la variable `x` à la valeur `v` et puis il devient `Q{v/x}`, tel que `{v/x}` indique la substitution de `x` par `v` dans le corps du processus `Q`. La communication des processus est synchrone, et les noms des canaux peuvent être envoyés et reçus comme étant des valeurs. Le processus `zero` représente l'inaction, les actions internes sont notées par `tau`, et la création d'un nouveau canal `z` dans un processus `R` est représenté par `(^z)R`, tel que la portée de `z` est limitée à `R`. Les processus peuvent être composés en parallèle par l'opérateur '`|`'. L'opérateur '`+`' est utilisé pour spécifier l'alternance ou le choix non-déterministe : `P+Q` est traduit par l'exécution de `P` ou `Q` mais pas les deux ensemble. Il y a aussi l'opérateur de correspondance utilisé pour la spécification du comportement conditionnel du processus : le processus `[x=z]P` est traduit par l'exécution de `P` si `x = y`, sinon il est traduit par le processus `zero`. La définition inductive de l'algèbre est la suivante :

$$\texttt{P} ::= \texttt{zero} \mid \texttt{tau} \mid \texttt{ch!(v).P} \mid \texttt{ch?(x).P} \mid \texttt{(\^{}z)P} \mid \texttt{[x=z]P} \mid \texttt{P+P} \mid \texttt{P|P}$$

Le choix local de processus est exprimé en combinant l'opérateur '`+`' avec des actions locales `tau`. Le processus `tau.P + tau.Q` est exécuté en tant que `P` ou `Q`. La règle principale de dérivation π- calcul est celle qui correspond à la synchronisation des services requis et fournis. Elle est décrite de la manière suivante :

$$\texttt{(}\cdots\texttt{ + x!(z).P + }\cdots\texttt{) | (}\cdots\texttt{ + x?(y).Q + }\cdots\texttt{)} \xrightarrow{\tau} \texttt{P | Q\{z/y\}}$$

La définition standard du π-calcul standard ne fournit pas le support des structures de données composées et les paramètres. La variante proposée par les auteurs supporte ses éléments. Par conséquent, les auteurs utilisent des notations spécifiques pour quelques structures de données comme les listes (`<>` et `++` pour la création et la concaténation) et les ensembles (`{}` et `++` pour la création et l'union). De plus, les auteurs autorisent l'utilisation des structures conditionnelles complètes par des choix non-déterministes :

$$\texttt{([G1]P1 + [G2]P2 + } \cdots \texttt{ + [Gn]Pn + [else]P0)}$$

La modélisation de l'interaction entre des composants CORBA est simple à modéliser avec cette variante de π-calcul puisque la manipulation des références d'objets et les invocations est soumise à la sémantique riche de l'algèbre de processus. L'approche de modélisation des composants et des interactions est basée sur les relations suivantes :

Référence d'une instance de composant	$\longmapsto$	un canal de communication π-calcul
Appel d'une méthode	$\longmapsto$	`ref!(m,(inputParams),(reply[..,excep,..]))`
Réponse aux appels de méthodes	$\longmapsto$	`reply!(returnValue,outputParams)`
Exceptions	$\longmapsto$	`excep!(excepParams)`

Une instance de composant est associée à un canal à travers lequel le composant reçoit les appels des méthodes. Ce canal est logiquement associé à la référence de l'instance. Une invocation de méthode `m` d'un composant dont la référence est `ref` est modélisée par l'action de sortie `ref!(m,(inputParams),(reply[..,excep,..]))` tels que `m` est le nom de la méthode, le tuple des paramètres d'entrée est `inputParams`, et `reply` est un tuple contenant le canal de retour et d'autre canaux de retour optionnels pour le traitement des exceptions. La réponse est représenté par un tuple envoyé par le composant appelé à travers le canal de retour reçu par la requête cliente, et contient la valeur de retour et les paramètres de sortie. `ref!(m,(args),(reply))` est réduit en `ref!m(args,reply)` ou en `ref!m(args)` si la référence du canal de retour est la même que celle du composant appelé. Le processus `ref?(m,(args),(rep)).[m='op']P` peut être réduit en `ref?op(args,rep).P`.

La spécification CORBA IDL de l'interface d'un composant Compte qui gère le compte bancaire d'un client peut être décrite par le code suivant :

```
interface Compte {
  exception CreditInsuffisant {float credit; float montant;};
  string debiter(in float amount) raises (CreditInsuffisant);
  string crediter(in float montant);
  float solde() ;
};
```

Le protocole d'un composant et défini par la construction `protocol` suivi par le nom de l'interface et la description textuelle du processus π-calcul qui représente le comportement en termes d'ordre d'appels et réception d'appels de méthodes. Le protocole du composant Compte peut être défini de la manière suivante :

```
protocol Compte {
  Compte (ref,credit) =
    ref?crediter(montant,rep).(^recu)rep!(recu).
    Compte(ref,credit+montant)
  +
    ref?debiter(montant,rep,crdInsuff).
    (
      tau.(^recu)rep!(recu).
      Compte(ref,credit-montant)
    +
      tau.crdInsuff!(credit,montant).
      Compte(ref,credit)
    )
  +
    ref?solde(rep).rep!(credit).
    Compte(ref,credit)
  +
    [else]
```

```
    Compte(ref,credit)
};
```

Le protocole du composant Compte décrit le comportement par un processus qui attend une requête qui arrive à partir du canal qui représente l'instance du composant. Dans le cas où l'opération est valide, le composant répond à la requête en envoyant un reçu `recu` au client ou provoque une exception et il revient à son état d'origine. Sinon, le processus ignore la requête. L'argument `credit` du processus est utilisé pour exprimer l'état interne du composant. Avec cette information supplémentaire, la spécification ne décrit pas seulement le protocole mais aussi sa sémantique permettant de décrire comment les trois méthodes modifient le solde.

D'autres approches de spécification des protocoles utilsent le π-calcul ont été proposées notamment celle qui est proposée dans [BBC05]. L'approche proposée est utilisée dans le contexte de l'adaptation logicielle (cf. section 2.5).

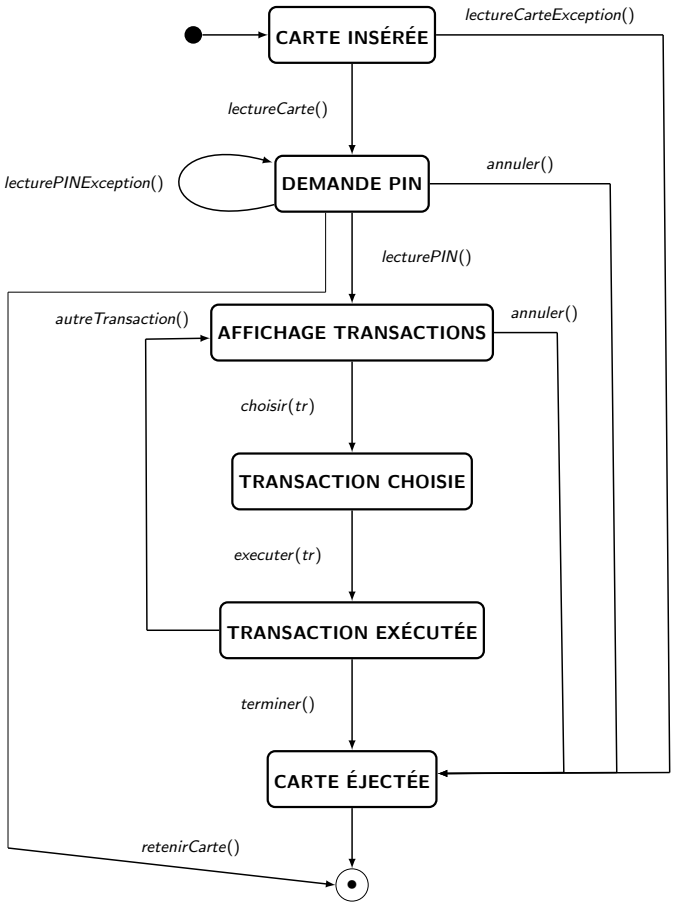


FIGURE 1.6 – Spécification *statechart* d'un distributeur bancaire de billets

1.5.2.3 Statechart et machines à états finies

Dans [MRR03], les auteurs proposent une approche qui vise à doter les composants UML par des modèles de spécification de leurs comportements pour spécifier et vérifier formellement leurs propriétés de sûreté. Le comportement d'un composant est décrit par un formalisme inspiré

des « *statecharts* de Harel » ou les « diagrammes d'état transitions » [Har87, OMG05] qui est représenté par un document annexe au diagramme de composants UML pour décrire le comportement. La figure 1.6 présente un exemple d'une spécification avec des diagrammes d'état transitions du *statechart* d'un composant qui gère le fonctionnement d'un distributeur bancaire de billets. Le formalisme proposé est un modèle mathématique pour décrire la connaissance relative au comportement des composants. Le modèle est une spécification hiérarchique de machines à états finies déterministes. L'approche est associée à un processus structurel qui supporte la simulation incrémentale, la vérification automatique, la génération de code ainsi que des tests pendant l'exécution. Le formalisme supporte des mécanismes de composition et de raffinement des machines à états hiérarchiques.

Les protocoles de comportement proposés par Frantisek Plasil et al. [PV02] sont des agents décrits par des expressions régulières. Cette sémantique définit des opérateurs de composition et de substitution d'agents. Les agents synchronisent sur des canaux typés en interfaces requises ou offertes partagées. Les auteurs distinguent trois niveaux de protocoles : (i) les protocoles associés aux interfaces décrivent les enchaînements licites au niveau d'une interface offerte ou requise ; (ii) les protocoles associés aux composants (appelés *frames*) décrivent les enchaînements licites aux niveaux de plusieurs interfaces ; (iii) les protocoles associés aux architectures décrivent les enchaînements licites au niveau d'une composition de composants.

Dans [PNPR05], les auteurs proposent une approche de description des protocoles des composants qui comble le vide entre les modèles formels utilisés pour la description des composants et leurs implémentations. L'idée de l'approche proposée est de décrire les protocoles des composants (implémentés en Java) par des systèmes de transition symboliques (STSs).

Dans [FGK03, Sö5, Pun99], les auteurs ont traité les protocoles des composants en utilisant les langages algébriques et non les expressions régulières pour modéliser la dynamique de communication entre eux. Un des intérêts de leur approche est de pouvoir spécifier des appels récursifs de services tout en étudiant les équivalences et le sous-typage. Les travaux de Schmidt [Sch03] portent sur les spécifications des composants fiables en incluant les contrats, les protocoles et les propriétés non-fonctionnelles ainsi que les éléments utilisés pour l'adaptation. Les protocoles sont définis par des automates au niveau des interfaces.

1.5.2.4 Automates

Dans [VZ05], Ivana Černá and al. ont proposé les automates CoIN (Component-interaction automata) est un formalisme pour la spécification et la vérification des systèmes à base de composants basée sur les automates. Le formalisme permet de spécifier les protocoles des composants dans les modèles de composants hiérarchiques. Chaque composant est associé à un automate. La composition des composants est traduite par la synchronisation de leur actions d'entrée/-sortie partagées. Chaque spécification d'un composant inclut la structure hiérarchique de ses sous-composants. Les composants primitifs sont identifiés par des identificateurs (entiers naturels) uniques et les composants composites par des identificateurs composés des identificateurs de leurs sous-composants primitifs.

Les automates d'interface (*Interface Automata*) [dAH01, AH05, AH01] ont été introduits pour modéliser les interfaces des composants. Un automate d'interface est un automate Input/Output [LT87] tel que les actions d'entrée ne sont pas nécessairement activables à tous les états de l'automate. Le protocole d'un composant est décrit par un seul automate d'interface. L'ensemble des actions est décomposé en trois sous-ensembles : les actions d'entrée, les actions de sortie et les actions internes. Les actions d'entrée représentent les méthodes ou les opérations implémentées par le composant et qui représentent ses services offerts. Elles peuvent aussi modéliser

les réceptions des messages dans un canal de communication. Les actions de sortie représentent les appels des opérations de l'environnement qui sont les services requis par un composant, les transmissions des messages dans un canal de communication et les exceptions qui se produisent pendant l'exécutions des méthodes. Le chapitre 3 contient plus de détails sur l'approche optimiste des automates d'interface. Les travaux présentés dans le cadre de cette thèse sont basés sur ce modèle d'où la motivation de consacrer tout un chapitre pour présenter en détail le modèle.

Dans [ASF+05], les auteurs proposent un formalisme basé sur l'approche optimiste des automates d'interface appelé automates d'interfaces sociables (*Sociable Interface Automata*). Le nouveau formalisme diffère légérement des automates d'interface au niveau de la sémantique de la synchronisation entre composants et le traitement de la concurrence. Plusieurs d'autres approches, basées sur les automates Input/Output pour la spécification des protocoles des composants, ont été proposées comme les travaux proposée dans [LNW06, AL03].

Après la présentation complète des concepts liés à la définition d'un composant en tant qu'une unité de composition avec d'autres au sein d'une application logicielle. Dans la section suivante, nous illustrons brièvement les concepts généraux liés à l'architecture suivant laquelle les composants sont assemblés pour construire l'application et la rendre utilisable.

1.6 Architectures logicielles

La notion de l'architecture logicielle est la base essentielle de toutes les technologies logicielles à grande échelle et elle est d'une importance capitale pour la conception des systèmes à base de composants et de leurs interfaces homme-machine IHM. Tout système logiciel complexe est basé sur une architecture qui peut être vue en termes de décomposition d'unités (composants et connecteurs) et d'autres éléments qui assurent les interactions entre ces différents unités à l'exécution. Ces interactions permettent de regrouper les différents composants du système pour construire des composants composites dans des styles architecturaux. La modélisation d'une architecture est basée sur plusieurs piliers. Nous allons en citer quelques une : la réglementation des interactions entre les composants et leur environnement, le rôle de chaque composant est défini d'une manière concise, et les interfaces IHM de l'application sont normalisées et standardisées à la fois pour les utilisateurs finaux et pour les assembleurs [Szy02]. Communément, l'architecture logicielle est axée sur la conception de la structure globale d'un système qui doit satisfaire les exigences fonctionnelles et non-fonctionnelles décrites dans le cahier des charges.

La définition fréquemment utilisée citée dans [BCK03] : « l'architecture d'un logiciel est définie par des *frameworks*[1] qui permettent d'implanter des composants, des connecteurs ainsi que les dispositifs qui permettent de relier entre eux et les déployer dans une plate-forme particulière. Les propriétés de tous ces éléments sont visibles de l'extérieur ». Donc, il est clair que l'architecture des logiciels et l'assemblage des composants sont les deux faces d'une même pièce, mais par ailleurs, l'architecture concrétise les propriétés et les fonctionnalités de tous les éléments qui construisent le système afin d'établir une description claire de son fonctionnement global. Plus concrètement, nous définissons le concept de l'architecture logicielle comme suit :

- une architecture logicielle se compose d'un ensemble de *frameworks* et un ensemble de *règles d'interopérabilité* entre les *frameworks* : une plate-forme est le support qui permet d'installer les *frameworks* de telle sorte que ces derniers puissent être instanciés, activés, et reliés ;
- un *framework* est une architecture de *composants composites* conçue selon un ensemble de

1. Un *framework* est l'ensemble des outils et des librairies nécessaires aux développements.

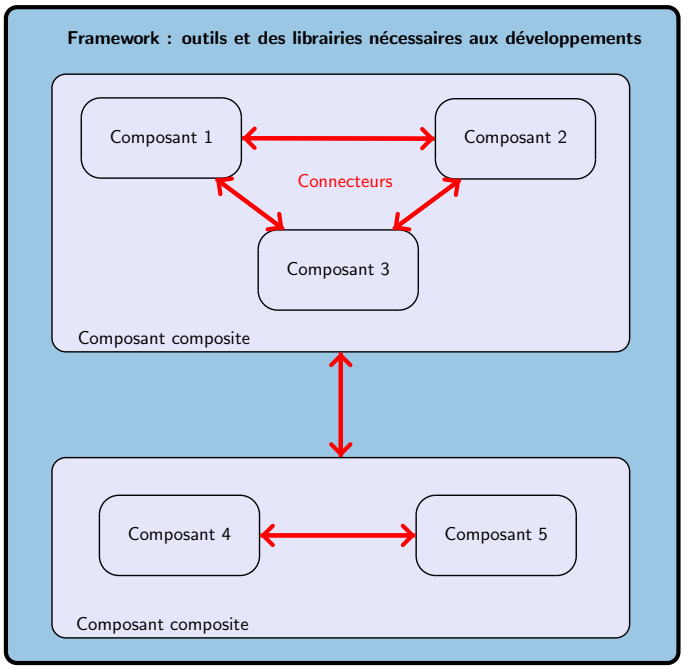


FIGURE 1.7 – Architecture logicielle : une représentation généralisée

mécanismes et *politiques* : les *frameworks* implémentent souvent les protocoles de communication entre les composants ;

- un *composant composite* est un ensemble de *composants primitifs* interconnectés et déployés simultanément ;
- un *composant primitif* est un ensemble de *classes* ou des *constructions non-orientées objet* (comme les procédures et les fonctions) et un ensemble de *ressources*.
- une *ressource* est une collection d'objets typés.

La figure 1.7 présente une vue généralisée de l'architecture logicielle.

1.6.1 Rôles

L'absence de l'architecture peut entraîner des problèmes sérieux lors de l'exécution et/ou la maintenance d'un logiciel. La modification du code d'un logiciel mal ou non architecturé peut déstabiliser sa structure et entraîne son *entropie* (dégradation). Le développeur constate que le logiciel est aussi complexe de telle sorte que celui-ci est extrêmement difficile à comprendre et à modifier. Le rôle de l'architecte est de prévoir le pire et de concevoir l'architecture la plus maintenable possible et la plus extensible possible [Pre00, Wik11].

La réutilisation des composants logiciels permet de réaliser les économies les plus substantielles à tous les niveaux. L'architecture logicielle permet donc une intégration harmonieuse des composants au sein de l'application. Le développement par réutilisation impose un cycle de création-réutilisation continu de composants et une architecture logicielle normalisée. La création de l'application est basée sur la conception d'une bibliothèque de composants qui facilite le développement de l'application. L'architecte doit explorer cette bibliothèque pour trouver les composants appropriés pour créer les composants manquants, les documenter et d'étendre la

bibliothèque avec. Une architecture doit satisfaire un ensemble de critères de qualité qui sont principalement [Wik11] :

- l'*interopérabilité* qui exprime la capacité d'un système logiciel de faire communiquer ses sous-composants et d'utiliser les ressources d'autres logiciels ;
- la *portabilité* qui exprime la possibilité de compiler et d'exécuter le système logiciel sur des plates-formes (machines, systèmes d'exploitation, environnements d'exécution) différentes ;
- la *validité* qui exprime le taux conformité des fonctionnalités du logiciel avec celles décrites dans le cahier des charges ;
- l'*intégrité* qui exprime la faculté d'un logiciel de protéger ses données des accès non autorisés ;
- la *fiabilité* qui exprime la capacité d'un logiciel de corriger ses propres erreurs de fonctionnement en cours d'exécution ;
- la *maintenabilité* qui exprime la simplicité de la maintenance du logiciel même durant son exécution ;
- la *réutilisabilité* qui exprime la capacité d'un logiciel de rendre réutilisables ses propres composants pour le développement d'autres logiciels ;
- l'*extensibilité* qui exprime la possibilité d'étendre simplement les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité ;
- la *transparence* qui exprime la capacité d'un logiciel de masquer à l'utilisateur les détails inutiles sur l'utilisation de ses fonctionnalités ;
- la *simplicité d'utilisation* qui décrit la facilité de l'utilisation d'un logiciel par les usagers.

Dans la sous-section suivante, nous présentons les langages de description d'architectures (ADL) utilisés pour décrire l'interopérabilité et l'assemblage des composants au sein d'une architecture logicielle.

1.6.2 Langages de description d'architectures (ADL)

Une architecture décrit l'ensemble des composants d'un logiciel et définit clairement les règles de leur assemblage. Elle représente le plan de construction d'un logiciel en se détachant des détails techniques de l'environnement de développement et en respectant le cahier des charges. En effet, la modification d'un plan est plus simple et moins couteuse que la modification d'une application déjà conçue.

Plusieurs travaux ont été menés par les communautés scientifiques et professionnelles pour répondre à ces besoins. Le résultat principal de ces travaux était de proposer des langages de description d'architecture (*Architecture Description Languages* ou ADL) [SG96, GP95]. Un langage de description d'architecture est « forme d'expression utilisée » pour la description des architectures [IF10]. Dans cette section, nous allons présenter les concepts généraux des principaux formalismes ADL existants.

Les concepts abstraits utilisés dans les langages ADL existants sont principalement au nombre de trois. La description des composants, les connecteurs, et la configuration ou la topologie. Les composants et les connecteurs sont assemblés pour former une configuration ou une topologie. La configuration structurelle de l'application est un graphe connexe des composants et des connecteurs. La configuration comportementale modélise l'évolution des liens entre les composants et les connecteurs, ainsi que l'évolution des propriétés fonctionnelles et non-fonctionnelles.

Les travaux sur les ADLs sont nombreux et ils ont émergé depuis le milieu des années 90. Ces travaux se concentrent sur la définition de langages déclaratifs qui sont classés en deux grandes familles [MT00]. La première correspond aux langages qui privilégient la description des éléments de l'architecture et leur assemblage structurel. Les langages de cette famille sont dotés d'outils de modélisation, d'analyseurs syntaxiques, et de générateurs de code. La seconde famille correspond aux langages de description de la configuration des architectures et sur la dynamique des systèmes. Ces langages sont associés à des plate-formes d'exécution ou de simulation en plus des outils de modélisation et de génération de code.

Parmi les langages les plus répondu, nous citons Darwin [MDEK95] qui est un langage de spécification des systèmes hiérarchiques à base de composants. Il supporte la description des structures dynamiques des composants évoluant durant l'exécution. Les primitives de base, sur lesquelles la sémantique de Darwin est construite, sont les composants et les services. Les composants dans Darwin sont construits d'un ensemble d'entités qui fournissent et demandent un ensemble de services.

```
component pipeline(int n) {
   provide output; require input;

   array F[n]: filter;
   forall k:0..n-1 {
       inst F[k] @ k+1;
       when k < n-1;
       bind F[k+1].input -- F[k].output;
   }
   bind F[0].input -- input; output -- F[n-1].output;}
```

L'exemple précédent montre une spécification Darwin d'un composant pipeline. Le nombre des sous-composants `n` est passé en paramètre. La sortie de chaque composant est reliée à l'entrée de son suivant (`bind F[k+1].input - F[k].output`). L'entrée du premier sous-composant et la sortie du dernier sous-composant sont reliées respectivement à l'entrée et la sortie du composant composite `pipeline` (`bind F[0].input - input; output - F[n-1].output`). Pour raisonner sur l'évolution des architectures, Darwin utilise le π-calcul [Mil99]. Le formalisme est idéal pour modéliser la reconfiguration dynamique des systèmes en cours d'exécution. Le système est modélisé par une collection de processus qui communiquent via des canaux de communication. Darwin décrit le comportement par Tracta [GKC99a] qui est basé sur les systèmes de transitions étiquetés (LTS). Une spécification est donnée pour chaque composant primitif et le comportement du composant composite est dérivé à partir des comportements de ses sous-composants en appliquant la composition parallèle entre les LTSs.

Le langage Wright [All97] (cf section 1.5.2.1) se distingue par les descriptions comportementales en terme de protocoles CSP. Rapide [RDT97] est distingué par la définition des simulations. Aesop [Gar95] est caractérisé par son pouvoir d'adaptation à plusieurs styles architecturaux. C2 [Med96] est adapté aux environnements répartis et évolutifs. Ce grand nombre de propositions a amené plusieurs classifications et comparaisons. La plus intéressante reste celle proposée dans [MT00]. Plusieurs méta-langages, tels que ACME [GMW97] ou ADML [Gro00] proposent une représentation intermédiaire exploitable par les outils architecturaux des différents ADLs pour permettre le passages entre eux.

De nos jours, les travaux de recherche sur les langages ADL changent de perspectives. Les travaux sur la modélisation des architectures et ceux concentrés sur la modélisation orientée objet, comme UML, restent complémentaires sans convergence constatée. Aujourd'hui, les travaux

comme ArchJava [ACN02] et l'intégration de la notion de composant dans UML 2.0 montre la fusion des deux approches. Le modèle de composants d'UML 2.0 permet de définir nettement l'architecture d'une application à base de composants provenant d'origines diverses. Le modèle représente un standard qui garantit les trois critères de qualité des systèmes ouverts à base de composants : la portabilité, l'interopérabilité, et la réutilisabilité.

1.6.3 Les styles architecturaux

L'architecture logicielle peut être catégorisée dans des styles divers. Les systèmes informatiques, construits au cours des dernières années, se classent parmi un nombre restreint de styles architecturaux. Un système peut utiliser plusieurs styles selon son niveau de granularité et son aspect [BCK03, GS94].

Un style architectural définit, d'une manière générique, un vocabulaire d'éléments, des contraintes de configuration de ces éléments, et une sémantique pour interpréter leur configuration. Dans cette sous-section, nous allons en citer quelques uns.

Modèles fonctionnels

Ce modèle d'architecture est basé sur la décomposition graduelle des fonctionnalités de l'application. Cette décomposition est fondée sur l'approche du raffinement graduel proposée par Niklaus Wirth [Wir71]. Elle consiste à découper une fonctionnalité en sous-fonctionnalités qui sont également divisées en d'autre sous-fonctionnalités et ainsi de suite. La fonctionnalité d'un module ou d'un objet est réalisée par des sous-modules ou des sous-objets. Une dérivation de cette architecture est l'architecture distribuée où les les modules ou les objets se retrouvent répartis sur un réseau.

Seeheim et Arch [TB85] sont des modèles d'architecture logicielle pour structurer l'interface homme-machine dans des applications interactives.

Modèles en couches

Le modèle en couches est la conséquence inévitable de la notion de la réutilisation des composants. En effet, les nouveaux composants utilisent les anciens. Une bibliothèque tend donc à devenir un empilement de composants. La division en couches consiste alors à regrouper les composants possédant des sémantiques semblables de manière à créer un empilement de paquetages de composants. Les composants des couches supérieures dépendants fonctionnellement des composants des couches inférieures et ainsi de suite.

Modèles orientés agents

L'architecture orientée agent représente la structure d'un système interactif composé d'une collection de composants spécialisés, appelés *agents*. Un agent est caractérisé par un état et une expertise et il est capable d'émettre et de réagir à des événements. L'agent est en contact direct avec l'utilisateur et il utilise de manière intelligente les autres agents pour réaliser ses objectifs. Il établit des dialogues avec les autres agents pour échanger de l'information. Plusieurs modèles orientés agents existent, nous citons à titre d'exemple les modèle PAC (*Presentation Abstraction Control*) [Cou87].

Il y a plusieurs autres styles d'architecture logicielle comme les architectures centrées sur les données (architectures 3-tiers et multi-tiers), les architectures basées sur la communication par flux de données (architectures de médiation et orientées évènements), etc.

1.7 Modèles à base de composants logiciels

Dans cette section, nous présentons une analyse des meilleures expériences actuelles de la technologie des composants. Trois approches principales sont examinées : CORBA/CCM de OMG, les *JavaBeans* et les EJBs de Sun Microsystems, et le modèle de composants Fractal de France Telecom et l'INRIA. Ainsi, nous présentons brièvement quelques autres modèles de composants utilisés pour faire le tour quasi complet sur l'utilisation des approches orientées composant dans l'industrie des applications logicielles à grande échelle.

1.7.1 EJB de Sun Microsystems

Dans le milieu professionnel, les applications logicielles gèrent généralement de gros volumes de données ce qui accroît leur niveau de robustesse. Par conséquent, la maintenance et la stabilité de ces applications sont des priorités pour les architectes et les développeurs. Le modèle d'architecture de base de ce genre d'applications est sans doute le modèle *3-tiers* qui est largement utilisé par les grandes entreprises ayant besoin de systèmes complexes basés sur la même organisation des informations. Ce modèle permet donc d'avoir plusieurs parties logicielles différentes et distribuées basées sur une même logique métier.

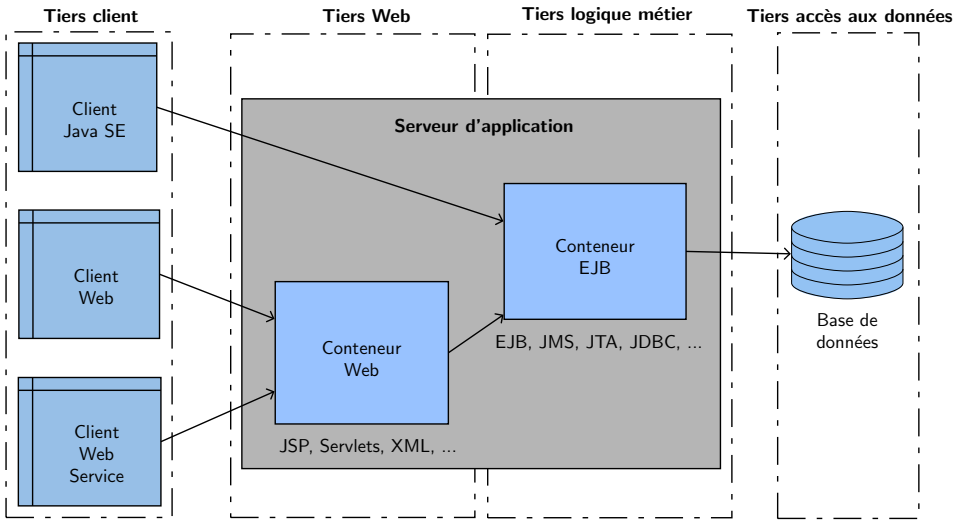


FIGURE 1.8 – Structure d'une architecture n-tiers d'une application en Java EE

Le modèle architectural 3-tiers est une évolution du modèle basique client/serveur. Il se compose de trois niveaux :

1. tier *client* qui correspond à la machine sur laquelle l'application client est lancée. Le tiers client utilise un navigateur pour représenter l'application sur la machine client ;
2. tier *métier* qui correspond à la machine sur laquelle l'application centrale est exécutée ;
3. tier *accès aux données* qui correspond à la machine gérant le stockage des données.

Toutefois, ce modèle de base est assez limité dans le cadre des architectures qui contiennent nombreuses entités. Par exemple, l'existence de plusieurs bases de données, interconnexion avec d'autres systèmes, l'existence de plusieurs clients, etc. L'architecture *n-tiers* généralisée est la plus adaptée pour prendre en compte la complexité évolutive de telles applications.

Le couplage de ces tiers est le problème crucial de cette architecture. C'est à ce niveau que les *frameworks*, comme Java EE (*Java 2 platform, enterprise edition*) [SM03], interviennent. Ils offrent un ensemble de librairies standards permettant d'interconnecter différentes technologies de développement. L'inconvénient principal de cette architecture concerne sa maîtrise. Son utilisation demande une très bonne aptitude à utiliser l'abstraction et les concepts basés sur l'utilisation des composants. La figure 1.8 présente la structure de l'architecture n-tiers des applications J2EE.

Le serveur d'application, qui représente les couches application et métier d'une application J2EE, est le médiateur entre la base de données et les clients. La couche application, représentée par le conteneur Web, est chargée de connaître et gérer l'état des sessions des clients connectés. Cette couche représente le contrôleur dans un modèle MVC [2] (*Model View Controller*) [Ree03]. Les éléments de base de cette couche sont souvent représentée par les *servlets* Java qui sont des pages JSP [3] (*JavaServer Pages*) compilées. Les servlets représentent les points d'accès à l'application. Elles doivent traiter les requêtes client et faire les appels nécessaires afin de récupérer ou d'enregistrer des données.

La couche métier est la couche principale de toute application. Elle est représentée souvent par un conteneur de composants *Entreprise JavaBeans* (EJB) [BMH06, SM96]. Elle doit s'occuper des accès aux différentes données et leurs traitements. Cette couche effectue la vérification de la cohésion entre les données et l'implémentation de la logique métier et les services de l'entreprise au niveau de l'application.

En dépit de quelques similarités, les EJBs, qui représentent principalement une technologie de composants côté serveur, ne doivent pas être confondus avec les composants *JavaBeans* ordinaires. Un composant *JavaBean* est une simple classe Java qui respecte un ensemble de conventions sur le nommage des méthodes, le constructeur, et le comportement. Ces conventions permettent la réutilisation, la substitution et la connexions des *JavaBeans* par des outils de développement. Un *JavaBean* doit hériter de l'interface `Serializable` pour pouvoir retenir et restaurer les états de ses instances. Il doit aussi rendre accessible publiquement ses attributs privés via des méthodes accesseurs.

1.7.1.1 Le modèle de composants EJB

Comme nous venons de l'évoquer, le modèle d'architecture distribuée induit le découpage d'une application en plusieurs parties. Cependant, ces applications ne peuvent donc pas être conçues à partir de rien. La création des plate-formes et les fragments logiciels de ce genre d'applications aurait un coût très important à tous les niveaux. Des standards ont alors vu le jour. Le plus connu est sans doute CORBA qui est le modèle idéal des applications distribuées. Néanmoins, la lourdeur et la complexité de ce modèle sont ses inconvénients majeurs. Le modèle EJB est plus restrictif, plus performant, et basé essentiellement sur Java, et a gagné une part de marché importante.

L'architecture EJB est composée d'au moins trois tiers : une machine cliente, une machine contenant la logique applicative, et une base de données. Les composants EJB implémentent la logique métier de l'application. Ils sont exécutés côté serveur contenus dans des conteneurs. Le serveur EJB joue le rôle d'un chef d'orchestre. C'est lui qui organise et synchronise les services et leur cycle de vie. Chaque partie de l'orchestre correspond à un service (EJB, transactions, etc). Tout les services sont indépendants mais ils travaillent ensemble pour produire un résultat

2. Modèle de conception séparant le modèle de données, l'interface utilisateur et la logique de contrôle.
3. `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`

commun. La figure 1.9 représente la structure interne, en termes de composants EJB et de services d'infrastructure, du tiers de la logique métier.

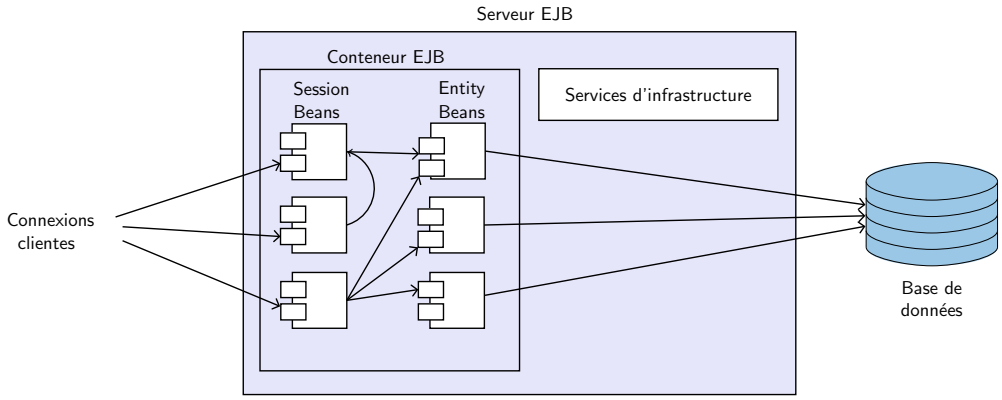


FIGURE 1.9 – Serveur EJB

Le conteneur EJB est un composant composite boîte noire qui gère la communication entre les EJBs et les services d'infrastructure disponibles au sein du serveur EJB, comme la connexion à la base de données, la gestion des transactions, la gestion de la sécurité, la gestion des cycles de vie des *beans* (composants dans le jargon Java), etc, en interceptant les appels aux *beans* qu'il contient. Ces services sont des méthodes de rappel (*callbacks*) fournis par les *beans*. Le serveur joue le rôle d'intermédiaire entre le système et le conteneur en permettant au conteneur de réaliser ses fonctions en faisant appel aux primitives du système. C'est le conteneur EJB aussi qui déploie les composants, les arrête et les redéploie.

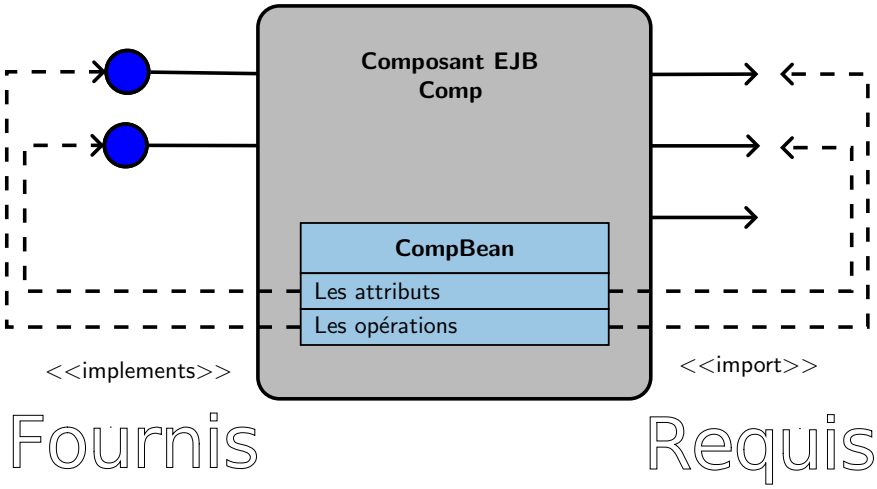


FIGURE 1.10 – Un modèle abstrait d'un composant EJB

1.7.1.2 Composants EJB

Les conteneurs EJB peuvent contenir trois types de *beans* : les *Entity Beans* qui représentent les objets persistants qui communiquent avec la base de données, les *Session Beans* qui fournissent

des séquences d'opérations pour les clients, et les *Message-Driven Beans* qui fonctionnent comme consommateur de messages asynchrones provenant des clients. Des contrats spécifiques sont définis pour chacun de ces types. La figure 1.10 représente un modèle abstrait de la composition générique d'un composant EJB.

Entity Beans

Les *Entity Beans* sont des composants persistants sous forme d'objets qui représentent des entités persistantes stockées dans un SGBD. La gestion de la persistance peut être faite par le *bean* lui même ou par son conteneur. Ces composants sont partagés par plusieurs clients concurrents et peuvent participer à des transactions. Ces composants résistent aux arrêts et aux pannes des serveurs EJB.

Session Beans

Les *Session Beans* sont des composants qui représentent un ensemble d'opérations pour les clients. Ces composants ne sont pas persistants et ils sont créés et tués par les clients. En contradiction avec les *entity beans*, ces composant ne résistent pas aux pannes ou à l'arrêt du serveur.

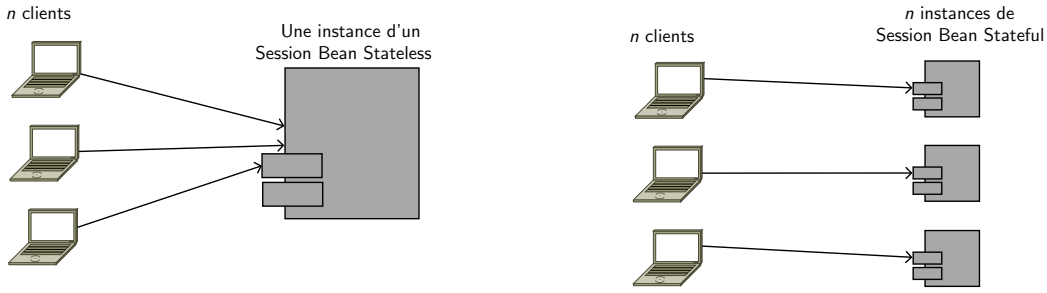


FIGURE 1.11 – *Stateless* vs. *stateful* session beans

L'état de ces composants est géré selon deux modes (cf. figure 1.11) qui sont décidés selon l'interaction désirée avec le client :

- les *stateless session beans* ("sans état") qui sont autonomes et ne dépendent pas d'un contexte particulier ou d'un autre service. Aucun état n'est conservé entre deux invocations de méthodes : lorsqu'une application cliente appelle une méthode, le composant exécute la méthode et retourne le résultat. L'exécution de la méthode ne dépend pas de ce qui c'est passé avant ou ce qui pourra passer après. La même instance d'un *bean* est utilisée par plusieurs clients.
- les *stateful session beans* ("avec état") concrétisent le concept de session entre le client et le serveur. Chaque client est lié à une instance de l'EJB et par conséquent, toutes les méthodes de l'EJBs sont partagées pour le compte d'un unique client. Un *stateful* conserve son état après une suite d'appels de méthodes.

Message-Driven Beans

Les *message-driven beans* fonctionnent comme des consommateurs d'événements asynchrones. Ce sont des composants non persistants utilisés pour la gestion des services des transactions. Ces

composants ne possèdent pas d'états et ils sont des consommateurs de messages JMS (*Java Message Service*) et l'accès au *bean* se fait en envoyant les messages vers la destination JMS qui s'agit d'une file d'attente qui stocke les messages.

La différence la plus visible entre les *message-driven beans* et les autres *beans* est que les clients n'accèdent pas aux *bean* à travers des interfaces. Un composant *message-driven bean* possède uniquement une classe qui représente un message.

Nous rentrons pas dans les détails de la gestion des cycles de vie des *beans* et les différences entre eux, mais nous allons illustrer comment le conteneur EJB fait cette gestion. Le conteneur permet aux clients de créer des *beans*, de les détruire, et de les rechercher. Le *bean* doit implémenter des *callbacks* (`ejbCreate`, `ejbPostCreate`, etc) pour assurer la gestion de son cycle de vie. Le conteneur fait aussi la gestion de l'état d'un *bean* en l'activant (réactivation du *bean* en chargeant l'état depuis la mémoire persistante) ou le passivant (désactivation du *bean* après l'enregistrement de son état dans la mémoire persistante).

1.7.1.3 Descripteur de déploiement

Le descripteur de déploiement est une spécification déclarative des propriétés d'un *bean*. Il est défini par le langage de description XML. Il permet de de définir les attributs de transaction, les champs persistants, l'environnement, la gestion ou pas par le conteneur, les éléments associés à la sécurité, etc. Chacune des propriétés est représentée par une balise. Par exemple, les attributs des transactions sont définis de la manière suivante :

```
<persistence-type>Container</persistence-type>
<container-transaction>
  <method>
    <ejb-name>CompBean</ejb-name>
    <method-name> ... </method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```

1.7.2 CORBA et CCM d'OMG

OMG (Object Management Group), fondée en 1989, est la plus large association d'entreprises de l'industrie informatique à but non lucratif. OMG fonctionne comme une organisation dont l'objectif est de standardiser, normaliser, et promouvoir le modèle objet de toutes ses formes. À l'origine, les efforts du groupe OMG étaient de résoudre le problème de comment les systèmes orientés objet distribués, implémentés dans des différents langages et exécutés sur différentes plate-formes, peuvent interagir ? Le résultat était CORBA (Common Object Request Broker Architecture) [OMG97] comme un premier fruit de travaux de ses membres.

Au début, l'objectif derrière CORBA était de rendre possible l'interconnexion ouverte d'une grande variété de langages de développement et de plates-formes logicielles, ce qui prolifère un niveau de portabilité plus large des applications. Principalement, CORBA est conçu de trois parties fondamentales :

1. un ensemble d'*interfaces d'invocation* ;
2. un *intergiciel ORB* (*Object Request Broker*) ;
3. un ensemble d'adaptateurs objet ;

L'invocation des méthodes objet est soumis au principe de l'expédition dynamique de méthodes (en anglais *dynamic method dispatch* ou *late binding*). Ce mécanisme permet de déterminer la bonne implémentation d'une méthode, dynamiquement durant le temps d'exécution. Les interfaces d'invocation autorisent plusieurs niveaux de liaisons dynamiques. Elles permettent aussi de transiter les arguments de la méthode invoquée à son implémentation de telle sorte que le noyau ORB puisse localiser l'objet receveur et la méthode. L'ORB est l'ensemble de classes et bibliothèques qui implémentent un "bus logiciel" par lequel les objets envoient et reçoivent des requêtes et des réponses, de manière transparente et portable. En d'autres termes, il s'agit de l'activation ou de l'invocation d'une des méthodes d'un objet distribué par un objet client. À la fin de la réception, un adaptateur objet invoque la méthode dans l'objet receveur. La spécification courante de ces adaptateurs est celle des POAs (*Portable Object Adapters*). Les clients appellent directement les objets par leurs références. Les références des objets sont crées par les POAs du côté serveur et puis elles sont passées aux clients.

Pour que ce processus puisse fonctionner, deux exigences doivent être satisfaites. Premièrement, toutes les interfaces doivent être décrites dans un langage commun. Deuxièmement, les liaisons entre les langages utilisés et le langage commun sont proprement définies. Ce langage commun, appelé *OMG Interface Definition Language* (OMG IDL), est une partie essentielle de CORBA.

1.7.2.1 Modèle de composants CORBA

Le modèle d'objet CORBA traditionnel, tel qu'il est défini dans CORBA 2.4 (OMG, 2000) et ses versions antérieures, a plusieurs limitations. Nous en citons quelques unes :

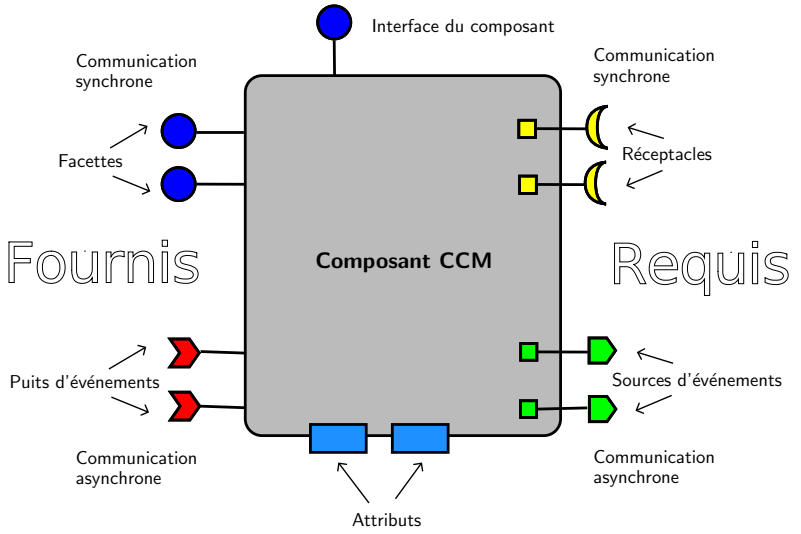
- la norme posait des difficultés au niveau de la mise en œuvre de la projection d'OMG IDL vers les langages de programmation ;
- pas de norme standard pour le déploiement des objets du côté serveur (uniquement des solutions *ad hoc*) ;
- pas de gestion standardisée des cycles de vie des objets ;
- pas de séparation claire entre les exigence fonctionnelles et non-fonctionnelles.

Tous ces inconvénients ont conduit à l'invention du modèle de composants CORBA CCM (*CORBA Component Model*) [OMG06]. CCM est inclu dans la version 3.0 de CORBA (OMG 2001). CCM étend le modèle d'objet CORBA en définissant des fonctions et services qui permettent aux développeurs d'implémenter, gérer, configurer, et déployer les composants qui intègrent les services CORBA couramment utilisés (tels que les transactions, la sécurité, des services de notification d'événements, etc) dans une architecture standard.

En outre, CCM permet de réutiliser plus de logiciels côté serveur et assure une configuration dynamique plus souple des applications CORBA. CCM propose toute une structure pour définir un composant, son comportement, son intégration dans un conteneur d'application, et son déploiement dans l'environnement distribué CORBA. Il est aussi important de noter que les composants CCM peuvent interagir directement avec les EJBs. L'architecture du modèle de composants CCM est formée de quatre éléments :

1. un *modèle abstrait de composants* décrit par une extention du langage IDL ;
2. un *modèle d'implantation* définit par OMG CIDL (*Component Implementation Description Language*) ;
3. un *modèle des conteneurs* qui définit la structure et l'interconnexion des composants par des conteneurs ;

4. un *modèle de déploiement* qui définit comment les composants et leurs conteneurs seront distribués, assemblés et déployés.




```
interface Facette_1 {
    // signatures méthodes
};

interface Receptacle_1 ;

component CCM{
    provides Facette_1 f1 ;
    ...
    uses Reseptacle_1 r1 ;
    publishes SEven_1 s1 ;
    consumes PEven_1 p1
};
```

FIGURE 1.12 – Composant CCM avec sa spécification IDL

1.7.2.2 Composants CCM

Les composants CCM sont les éléments de base dans un système CCM. Une contribution importante de CCM découle de la standardisation du cycle de développement des composants en utilisant CORBA comme étant son infrastructure middleware. Les développeurs de composants CCM doivent définir les interfaces IDL de chaque composant. Ensuite, ils implémentent les composants. Les implémentations des composants résultantes peuvent être alors rassemblées dans des fichier d'assemblage, comme les librairies partagées, des fichiers JAR, ou des fichiers DLL et liés dynamiquement. Finalement, un mécanisme de déploiement est utilisé pour déployer les composants dans un serveur d'applications qui héberge leurs implémentations en chargeant leurs fichiers d'assemblage. Ainsi, les composants s'exécutent dans des serveurs d'applications et sont disponibles pour traiter les demandes des clients. La figure 1.12 présente le modèle abstrait d'un composant CCM et sa définition IDL.

Les attributs d'un composant CCM sont ses propriétés configurables indispensables pour sa réutilisation et sa configuration. Les attributs peuvent être configurés par les mécanismes de feuilles de propriétés utilisées pour l'assemblage et la configuration. En contradiction avec les anciennes versions de CORBA, CCM permet aux opérations qui manipulent ses attributs de produire des exceptions. Le développeur du composant peut soulever une exception dans le cas où une tentative de changement des attributs est faite après la configuration complète du système.

Un composant CCM utilise aussi un ensemble de ports pour communiquer avec son environnement. Ces ports améliorent considérablement la réutilisation des composants par rapport au modèle CORBA traditionnel. Les ports d'un composant CCM sont classifiés en plusieurs catégories :

– les *facettes* ou les interfaces fournies par le composant. Elles sont invoquées d'une manière synchrone ;
– avant qu'un composant puisse invoquer les opérations d'un autre composant, il doit d'abord créer une connexion avec l'instance de ce dernier en acquérant sa référence. Les ports des ces connexions sont appelés *réceptacles*. Ils représentent les interfaces requises par le composant ;
– des événements asynchrones peuvent se produire durant la communication entre les composants. Ces interactions par les événements sont basées sur le modèle de conception (*design pattern*) observateur. Un composant déclare son intérêt de publier ou de consommer des événements en précisant ses *sources d'événements* et les *puits d'événements* dans sa définition.

Pour standardiser le cycle de vie d'un composant, CCM introduit le mot clé IDL *home* qui spécifie les stratégies prises pour structurer les cycles de vie d'un composant. Chaque interface *home* est spécifique à son composant et les clients peuvent accéder à ces interfaces pour contrôler les cycles de vie des composants avec lesquels ils communiquent. Par exemple, l'interface *home* peut créer et tuer des instances de composants.

1.7.2.3 Conteneurs CCM

Les composants CCM dépendent de la norme POA pour expédier les requêtes clientes entrantes. Toutefois, contrairement aux versions précédentes de CORBA, le développeur de l'application n'est plus responsable de la création de la hiérarchie POA. Le modèle de composants CCM utilise la spécification du composant pour configurer automatiquement la hiérarchie POA et localiser les différents services définis par CCM. De plus, les composants peuvent exiger la communication via des fonctions de rappel (*callbacks*) lorsque certains événements se produisent. Pour prendre en charge les fonctionnalités décrites ci-dessus, le serveur de composants instancie des *conteneurs*. Le modèle de programmation des conteneurs CCM définit un ensemble d'interfaces API qui simplifie la tâche de développement et de la configuration des applications CORBA. Un conteneur encapsule les implémentations des composants et fournit un environnement pour les exécuter.

1.7.3 Fractal de France Telecom & l'INRIA

La technologie à base de composants Fractal [BCL$^+$04, BCL$^+$06] fournit un modèle hiérarchique de composants imbriqués pour le développement des systèmes logiciels complexes. C'est un modèle général d'implémentation, de déploiement, et de gestion (surveillance, contrôle, et configuration dynamiquement) des composants. Il a été développé par France Telecom R&D et l'INRIA au sein du consortium ObjectWeb, Le modèle Fractal dispose aussi d'un canevas logiciel qui est un ensemble d'APIs indépendants de toute implémentation et qui peut servir comme étant une base à diverses extensions. Fractal a été fondé pour résoudre des problèmes posés par les modèles de composants standards, comme par exemple l'expressivité restreinte de ces modèles qui ne permettent pas modéliser proprement la composition hiérarchique et le partage des composants, l'adaptation, et la reconfiguration.

1.7.3.1 Modèle abstrait

Le modèle abstrait Fractal est composé d'un ensemble de composants qui communiquent par transmission de signaux nommés via des interfaces. En plus des interfaces standard permettant

l'accès aux fonctionnalités des composants, les composants Fractal disposent d'un ensemble d'interfaces de commande et des contrôleurs permettant de gérer le cycle de vie des composants (démarrage/arrêt des composants, le paramétrage des attributs). Ces contrôleurs permettent aussi de changer la structure interne des composants en ajoutant ou supprimant des sous-composants et les liaisons entre eux. Le contrôleur représente la structure d'un composant, il permet d'intercepter les signaux entrants et sortants du composant et il peut modifier aussi le comportement de ses sous-contrôleurs. Le contrôleur assure aussi la reconfiguration dynamiquement du composant et la gestion des aspects techniques non-fonctionnelles comme la gestion des transactions et la sécurité.

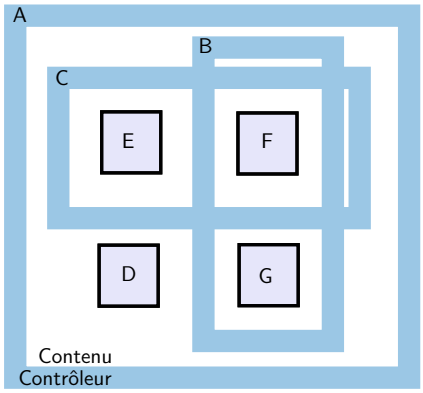


FIGURE 1.13 – Modèle de composants Fractal

Un composant Fractal est composé de deux parties : le contrôleur (membrane) et un contenu. La membrane encapsule le contenu et contrôle les requêtes entrantes traitées par le contenu. Toutes les requêtes adressées au composant sont enfilées dans un buffer tampon à l'intérieur de la membrane et traitées d'une manière FIFO. Un composant peut être imbriqué dans un composant englobant comme étant un sous-composant non partagé ou partagé avec d'autres. Par exemple, dans la figure 1.13, le composant F est partagé entre les composants B et C. Le partage des composants modélise le partage de leurs ressources, leurs activités, et leurs domaines de contrôle. Les composants peuvent être, par conséquence, composites ou primitifs. Les composants primitifs encapsulent du code exécutable et ils peuvent servir à encapsuler du code déjà existant (*legacy*). Le contenu des composants composites encapsulent un ensemble de sous-composants. Les composants Fractal peuvent être instanciés selon deux modes : les modes *Factory* et *template*. Le mode *Factory* crée n'importe quel type de composant. Le mode *Template* crée des composants clones à partir d'autres composants.

Un composant Fractal dispose d'un ensemble d'interfaces requises et fournies. Les interfaces fournies représentent les services réalisés par le composant. Les interfaces requises décrivent les services dont le composant a besoin pour réaliser ses propres services. Chaque interface (fournie ou requise) a son homologue interne utilisé pour la connexion du composant avec ses sous-composants. L'interface homologue interne a un type opposé à son équivalente externe. C'est-à-dire, pour chaque interface fournie externe, il existe une interface requise interne homologue connectée à une interface fournie d'un sous-composant (délégation) et vice versa (subsumption).

Un composant Fractal dispose aussi d'un ensemble d'interfaces de contrôle qui fournissent des services permettant de gérer le composant. Elle permettent de contrôler les connexions des sous-composants et de configurer leurs cycles de vie (ajout/suppression, suspension/reprise d'exé-

cution, instanciation des liaisons, etc). Chaque interface a un nom et un type qui représentent ses références. Une interface peut être déclarée optionnelle. Une interface optionnelle peut ne pas être liée à une autre interface. Elle peut être marquée comme étant multiple et dans ce cas elle contient plusieurs sous-interfaces.

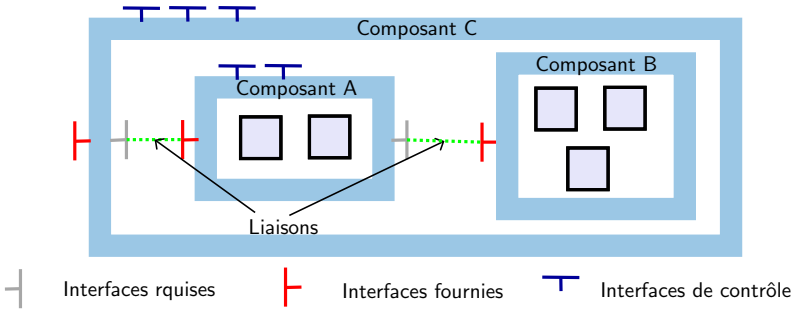


FIGURE 1.14 – Interfaces des composants Fractal

Les liaisons (*bindings*) entre les composants s'effectuent entre les interfaces et elle sont de deux types : des liaisons simples et composées. Les liaisons simples relient exactement deux interfaces locales au sein d'un même composant. Les liaisons composées permettent la communication d'un nombre arbitraire de composants quel que soit le type (fournies ou requises) de leurs interfaces. Les composants de liaison sont appelés aussi connecteurs. La figure 1.14 représente les interfaces de composants Fractal et les liaisons entre elles.

Fractal dispose d'un langage de description architecturale sous la forme de descripteurs XML « *Fractal ADL* ou Fractal Architecture Description Language ». Ce langage spécifie les types de composants et leurs ports, le instances de composants, les contrôleurs des composants ainsi que leurs compositions. La composition est effectuée par construction d'une hiérarchie d'instances de composants liés. Le concepteur peut donc personnaliser le contrôle des composants à sa convenance.

1.7.3.2 Implémentations

Julia

Le modèle Julia [Bru03] est une implémentation Fractal de référence en Java et encore en cours de développement. Il représente un *framework* extensible et configurable pour programmer des composants Fractal de différentes formes. La spécification des composants Fractal dans Julia a été étendue en ajoutant la spécification de leurs comportements par des protocoles de communication [AP05].

Fractive/ProActive

Fractive [BCM03] est une implémentation Fractal qui utilise l'intergiciel ProActive [CKV98] pour la distribution. Les traits caractéristiques de ProActive sont les appels asynchrones des méthodes, l'absence du partage de mémoire, la transparence de la distribution, et la migration.

ProActive est une implémentation Java des objets distribués dont les appels de méthodes sont asynchrones exploitant des références futures. Une référence future est une référence à la valeur de retour d'un appel de méthode qui n'est pas prêt d'être exécuté mais il va éventuellement être évalué. Cette référence est mise à jour après par la valeur de retour. Le système ProActive

est composé par plusieurs composants d'activité. Chaque composant définit son point d'entrée appelé objet actif qui est référencé de l'extérieur et exécuté dans un processus à part entière. Les objets passifs ne peuvent pas être référencés directement de l'extérieur de leur composants et ne peuvent pas être exécutés dans des nouveaux processus.

1.7.4 Autres modèles

Il existe d'autres modèles de composants utilisés pour développer des applications logicielles. Nous allons présenter les plus répandus.

Modèle de composants .NET de Microsoft

Microsoft regroupe l'ensemble de ses produits de développement et d'exécution des applications réparties à base de composants dans la plate-forme .NET. .NET [Mic] offre le support de plusieurs langages tels que C++, VB.NET, et C# (C Sharp). Tous ces langages supportés peuvent être compilés en un code intermédiaire MSIL (*Microsoft Intermediate Language*). Ce code intermédiaire est exécuté par le CLR (*Common Language Runtime*). De plus, .NET met l'accent sur la réutilisation des (*Web Services*). Ainsi, WSDL (*Web Service Definition Langage*) est un format de représentation des interfaces de Web Service en XML.

Les technologies Microsoft orientées composant COM, DCOM, MTS, et COM+ sont conçues pour la distribution de composants dans des environnements répartis. COM repose sur des conventions d'interopérabilité binaire entre les composants et sur les interfaces. DCOM intègre la distribution dans COM, et MTS étend DCOM avec les services de persistance et de transaction. L'ensemble constitue COM+.

Modèle de composants OSGI de OSGi Alliance

Le modèle de composants OSGI a été fondé en 1999 par le groupe OSGi Alliance pour la création d'une spécification ouverte pour la délivrance de services multiples à partir du net aux réseaux locaux [OSG]. OSGI représente un *framework* léger qui peut être exécuté dans les dispositifs matériels à mémoire réduite. OSGI s'appuie sur Java pour assurer la portabilité sur différents matériels. Une caractéristique importante de cette technologie est le fait qu'elle a été adaptée à l'évolution dynamique des architectures système. Les composants peuvent être téléchargés, mis à jour, et supprimés dynamiquement sans arrêter le système. En outre, OSGI permet l'administration à distance des systèmes via le réseau.

OSGI est basé sur deux concepts principaux qui peuvent être interprété comme étant des composants : paquets (*bundles*) et services. Le développeur fait le design de l'application en créant un ensemble de paquets qui contiennent des services. Un service implémente un ensemble de fonctionnalités. Un service est une unité de composition défini par son implémentation et un ensemble d'interfaces. Un système est un ensemble de services coopératives et connectés. Un paquet est une unité de déploiement qui regroupe un ensemble de services. Les paquets sont téléchargés après sur demande [OSG]. Les composants sont associés aux concepts utilisés dans l'application.

1.8 Synthèse

Dans le cadre de ce chapitre, nous avons détaillé les termes et les concepts liés aux composants logiciels en définissant une ontologie complète sur les relations entre eux. Ensuite, nous

avons introduit les définitions des concepts reliés aux interfaces des composants et comment la sémantique de leurs opérations est intégrée pour définir leurs contrats. De plus, nous avons défini les protocoles comportementaux des composants et les différents langages utilisés, dans la littérature, pour les spécifier. Nous avons également illustré brièvement les architectures logicielles des systèmes à base de composants. À la fin du chapitre, nous avons présenté un panorama des fondements des modèles de composants les plus utilisés comme EJB, CCM, et Fractal, ainsi qu'une brève présentation des modèles .NET et OSGI.

Chapitre 2

Assemblage et interopérabilité des composants

Dans le premier chapitre, nous avons présenté une ontologie sur les termes et les concepts liés aux composants logiciels. Nous avons détaillé les relations entre les notions des interfaces et les spécifications contractuelles des composants ainsi que leurs protocoles comportementaux. Ces notions ont été introduites pour définir le composant logiciel en tant qu'une identité parmi d'autres dans les systèmes modernes. Dans ce chapitre, nous déplaçons la loupe pour examiner les caractéristiques des relations entre le composant et ses semblables dans un environnement logiciel. Nous expliquons de manière générique et indépendante de tout contexte spécifique, comment l'assemblage et l'interopérabilité des composants sont mises en œuvre en utilisant leurs interfaces.

Dans la section 2.1, nous présentons brièvement le processus d'assemblage. Ensuite, nous rentrons plus dans les détails techniques de l'interopérabilité des composants en termes de demande et d'acquisition de services. Dans la section 2.2, nous présentons les bases conceptuelles et les outils permettant de spécifier les connecteurs des composants. Dans la section 2.3, nous explicitons les différents niveaux de vérification de l'interopérabilité des composants. Dans la section 2.4, nous faisons un tour d'horizon sur les travaux qui concernent la vérification de la sûreté de l'assemblage des composants au sein d'un logiciel. Dans la section 2.5, nous explicitons la notion d'adaptation des composants comme étant un cas particulier de l'assemblage des composants incompatibles.

2.1 Opération d'assemblage

Comme nous avons introduit avant, un composant est une unité de réutilisation destinée à communiquer avec d'autres. Dans ce contexte, l'étape de composition ou d'assemblage des composants, au sein d'une application logicielle, est de nécessité fondamentale et doit être bien fondée pour assurer la cohérence entre les différentes fonctionnalités de l'application. L'assemblage des composants est basé essentiellement sur deux axes : (i) la spécification de l'interconnexion entre deux ou plusieurs composants et (ii) la substitution d'un composant avec un autre plus évolué dans un système sans perturber son fonctionnement. Le deuxième fondement est indispensable car il est en relation directe avec les critères reliés à l'évolution du système. La substitution d'un composant C par un composant C' est dit sûre si tous les environnements qui fonctionnent avec C fonctionnent aussi avec C'.

L'opération de l'assemblage relie les composants par des connecteurs. La notion de connecteur

recouvre l'ensemble des moyens nécessaires pour interconnecter les composants en synchronisant les sorties (services requis) des uns avec les entrées (services offerts) correspondantes des autres. Ces moyens peuvent être distingués en deux catégories déterminées selon les contraintes et le contexte dans lesquels les composants sont induits. La première catégorie est l'utilisation des spécifications contractuelles des composants et leurs propriétés pour modéliser les connecteurs des composants d'une manière implicite. Un connecteur est vu comme un modèle de communication entre deux ou plusieurs composants basés sur la spécification de leurs contrats. Cette interopérabilité peut être vérifiée au stade de trois niveaux [Weg96] :

1. le niveau *signature* dont l'objectif est de s'assurer que les signatures des opérations sont compatibles. La signature de l'opération dans le composant appelant doit être la même que l'opération appelée. Les deux signatures doivent avoir le même nom, le même nombre de paramètres, et les paramètres doivent avoir les mêmes types en respectant leurs ordres dans les deux signatures ;
2. le niveau *protocole* dont l'objectif est de s'assurer que les protocoles des composants sont compatibles ou non ;
3. le niveau *sémantique* dont l'objectif est de s'assurer que les sémantiques des opérations partagées, modélisées souvent par des pré et post-conditions, sont compatibles.

Dans des cas particuliers, la deuxième catégorie des connecteurs se manifeste comme étant la spécification et le développement des moyens logiciels ou matériels explicites pour assurer l'assemblage des composants. Ce processus est appelé l'*adaptation* des composants. Il est utilisé dans le cas où l'interopérabilité implicite des composants n'est pas garantie par moyen de leur contrats comportementaux à cause des disparités entre les descriptions des services. En effet, des incompatibilités possibles puissent se produire entre les interfaces aux différents niveaux de composition. Dans ce cadre, la séparation des tâches est très importante. Il est souhaitable de ne pas faire apparaître les attributs des connecteurs dans la spécification des composants mais d'en faire des entités bien distinctes [ACC02]. D'autre types de composants intermédiaires sont utilisés pour connecter un ensemble de composants dans un réseau de diffusion de messages ou d'événements. Ces composants sont appelés canaux ou groupes de diffusion entre des composants sources et puits d'événements.

Le déploiement des composants dans une plate-forme d'accueil est indispensable pour que l'assemblage soit fonctionnel. La plate-forme d'accueil correspond à l'environnement système dans lequel les composants sont installés, instanciés et activés. Il peut s'agir d'un système d'exploitation ou d'un environnement dédié à un certain modèle de composant (cf. section 1.6).

2.2 Connecteurs de composants

Plusieurs approches de spécification ont été fondées pour définir les connecteurs des composants. Au niveau de l'approche orientée objet, les connecteurs sont basés sur les dépendances des appels des méthodes des interfaces d'objet configurés en cours de l'exécution. Les références des objets sont utilisées pour localiser l'implémentation des méthodes invoquées. Les mécanismes de connexion sont devenus des services à part entière. Par exemple, les services de transmission de messages et d'échange d'événements qui sont la base de tout intergiciel de communication basé sur la distribution de messages (*Message-Oriented Middleware* MOM) [Szy02].

Les approches orientées connexion deviennent nécessaires dans le cas ou la construction de l'application est basée sur l'utilisation des composants préfabriqués. Les composants doivent être vus comme des entités entourées par des ports de connexion entrants et sortants. Les interfaces

qui correspondent à ces ports sont appelées interfaces fournies ou requises. Une interface fournie correspond souvent à une opération ou une méthode offerte. Les appels des opérations sont acheminés à travers les interfaces requises.

Les paramètres des opérations des interfaces fournies peuvent être utilisés pour les valeurs de retour. A ce niveau une confusion peut avoir lieu entre les interfaces requises et les valeurs de retour. La valeur de retour de l'appel d'une méthode et la variable dans laquelle est enregistrée dans le composant appelant portent le même type d'objet. Les interfaces requises dans l'approche orientée objet représente les interfaces distantes dans lesquelles les méthodes appelées sont déclarées. Dans d'autres normes, les interfaces de sortie correspondent à déclarer les évènements ou les messages qu'un composant peut émettre. L'envoi d'une valeur de retour est représenté par une "action" de sortie. Par conséquent, les interfaces sont fournies ou requises selon les caractéristiques des composants et leurs cadres d'utilisation.

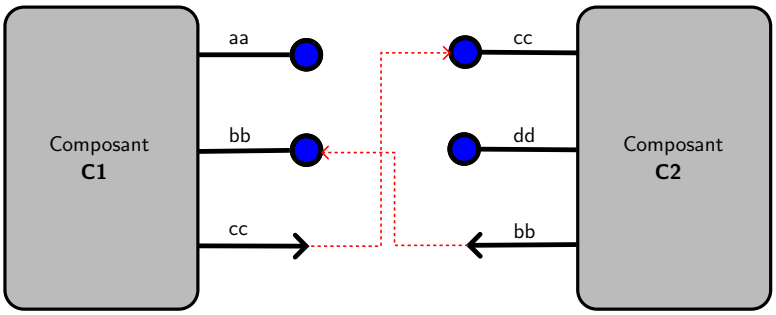


FIGURE 2.1 – Interfaces fournies et requises

Comme le composant appelé n'a aucune information sur le composant appelant, le composant appelant doit donc tenir une référence explicite ou un lien statique vers le composant appelé. Les interfaces dont un composant dépend sont tout simplement importées. La figure 2.1 illustre la connexion entre deux composants par des "câblages symboliques" entre les interfaces d'entrée et de sortie. L'interface `cc` est requise par le composant C1 tandis que les interfaces `aa` et `bb` sont fournies. Au contraire, l'interface `bb` est requise par le composant C2 et il fournit les interfaces `cc` et `dd`. Les connexions sont établies entre les interfaces `bb` et `cc`. Ces connexions sont indépendantes des concepts d'importation et d'exportation.

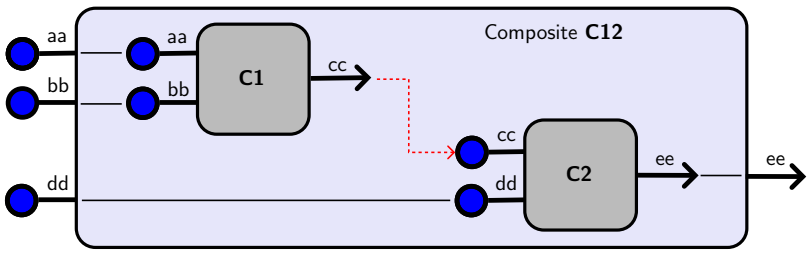


FIGURE 2.2 – Composant composite de C1 et C2

Dans d'autres approches à base de composants, comme les EJBs, il y a une séparation claire entre les interfaces requises et fournies. En Java, les interfaces requises sont importées par des `import` et les interfaces fournies sont représentées par le mot clé `implements`. Plusieurs approches

à base de composants regroupent les instances des composants connectés en composants composites. La figure 2.2 montre un composant composite C12, une fois instancié, il crée deux instances des sous-composants C1 et C2 et les connecte par le couplage de leurs interfaces fournies et requises [Szy02].

Si on considère que les connexions entre les composants sont des entités explicites à part entière, d'autre réseaux de connexion entre les composants viennent à l'esprit. Par exemple, un composant appelant pourrait être relié à un ensemble de composants appelés par des réseaux de diffusion. D'autre notions s'imposent comme les canaux des groupes de communications afin que les composants "sources" d'événements communiquent avec un ensemble de composants "puits". Par conséquent, des composants puits peuvent être ajoutés ou supprimés d'un groupe sans perturber la communication. Les composants CCM communiquent de cette manière via leurs interfaces sources et puits d'événements (cf. section 1.7.2.1).

Les services des canaux ou les groupes peuvent être utilisés pour découpler les connexions entre les composants sources et puits d'événements. Le canal, comme c'est présenté dans la figure 2.3, est représenté par un composant à part. Ce composant permet de relayer les appels passés via les mêmes méthodes ou procédures, mais qui portent des messages différents. Ces composants peuvent être utiles aussi pour réaliser d'autres services qui permettent de rendre la communication plus fiable. Un canal, par exemple, peut filtrer, compter, retarder, ou enregistrer les appels pour les exécuter plus tard.

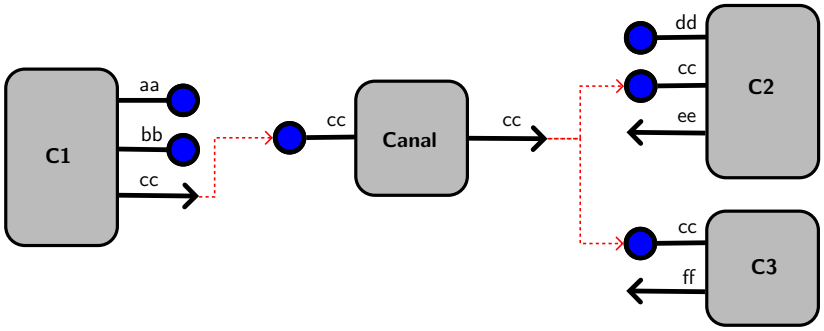


FIGURE 2.3 – Séparation du canal d'événements des composants sources et puits

2.3 Niveaux d'interopérabilité des composants

Le développement à base de composants a orienté l'ingénierie logicielle du développement ordinaire au développement par assemblage de composants préfabriqués comme c'est déjà mentionné précédemment. Actuellement, les efforts de développement s'orientent plutôt vers la découverte progressive des composants logiciels et leurs critères fonctionnels, les prévisions (hypothèses) sur leur fonctionnement interne, et les incompatibilités qui peuvent avoir lieu lorsqu'ils sont utilisés. Par conséquent, l'une des principales questions de la conception des logiciels à base de composants réutilisables est l'*interopérabilité*.

L'interopérabilité peut être définie par la capacité de deux ou plusieurs composants de communiquer et coopérer malgré les différences de leurs langages d'implémentation, leurs environnements d'exécution, et leurs modèles d'abstraction [Kon95, Weg96, BS00]. Traditionnellement, deux principaux niveaux d'interopérabilité ont été distingués : le niveau *signature syn-*

taxique(noms et signatures des opérations) et le niveau *sémantique* (le sens d'une opération). Le premier traite l'assemblage en termes de "prises mâles et femelles" et le second traite les aspects comportementaux de l'interopérabilité des composants. Dimitri Konstantas décrit ces deux niveaux par les termes "statique" et "dynamique" dans [Kon95].

L'interopérabilité syntaxique est bien définie et élaborée. Les architectes des logiciels ont donné naissance aux plate-formes à base de composants comme CCM, EJB, ou COM/DCOM. L'interopérabilité n'est pas suffisante pour assurer le développement correct et sûr des systèmes ouverts. D'autre part, d'autres propositions s'intéressent au niveau sémantique et proposent d'enrichir les interfaces des composants par la description du comportement en spécifiant la sémantique des opérations [LS00]. Le traitement de la sémantique comportementale des composants, qui est beaucoup plus puissant que la description du niveau signature, introduit des difficultés sérieuses dans les applications à grande échelle. La complexité de prouver les propriétés comportementales des composants, qui exige l'utilisation des méthodes formelles, entrave sa mise en œuvre dans la pratique. L'utilisation des méthodes formelles dans l'industrie pour vérifier la correction des applications est encore très réduite. Il y avait quelques contributions à ce niveau dans les milieux académiques et professionnels, mais nous sommes encore loin d'avoir des solutions satisfaisantes.

Dans le cadre de la spécification comportementale des composants, un autre niveau d'interopérabilité doit être pris en compte. C'est le niveau protocole qui traite seulement l'ordre partiel entre les méthodes et les opérations communicantes des composants et les conditions de blocage qui empêchent la disponibilité des services des composants. Ce niveau d'interopérabilité, introduit dans [YS97], fournit un mode plus puissant et rigoureux de la vérification de l'interopérabilité entre composants logiciels.

Les sections 1.4 et 1.5 du premier chapitre présentent les niveaux signature, sémantique, et protocole dans le cadre de la spécification des interfaces des composants. Dans cette section, nous allons plutôt examiner l'exploitation de ses trois niveaux dans le cadre de l'interopérabilité et la vérification de la compatibilité entre les composants.

2.3.1 Niveau signature

Au niveau signature, l'interopérabilité des composants est basée sur les signatures des opérations. La signature d'une opération est représentée par son nom, les paramètres, et les valeurs de retours. Deux principaux niveaux de vérification peuvent être effectués entre les composants : la *compatibilité* et la *substitution*.

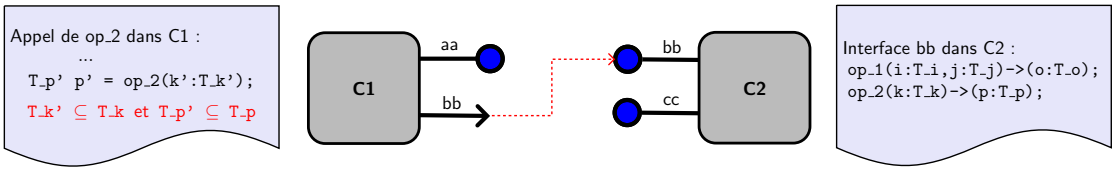


FIGURE 2.4 – Compatibilité des composants au niveau signature

La compatibilité peut être décrite par la capacité de deux instances de composants connectées de fonctionner correctement. Les plate-formes de composants (comme DCOM, CCM, EJB, etc) fournissent déjà des mécanismes pour assurer une interopérabilité des composants. Ces plate-formes assurent la communication entre des composants hétérogènes basée sur des accords syntaxiques. Ils utilisent les spécifications IDLs des composants pour donner les informations

syntaxiques sur leurs services, indépendamment de leurs langages et de leurs implémentations, pour assurer leurs communication.

La signature de l'appel d'une méthode dans le composant appelant doit être compatible avec la signature de la méthode dans l'interface fournie par le composant appelé. Cette compatibilité est traduite par la vérification si la signature de l'appel et celle déclarée dans l'interface fournie ont le même nombre des paramètres d'entrée et de sortie (valeur de retour) et que la règle de sous-typage, décrite dans la figure 2.4, est satisfaite. L'appel `T_p' p' = op_2(k':T_k')` de la méthode `op_2` dans le composant C1 est syntaxiquement compatible avec la déclaration de l'opération `op_2(k:T_k)->(p:T_p)` dans le composant C2 si (i) le nombre de paramètres et le même et (ii) `T_k'` $\subseteq$[1] `T_k` et `T_p'` $\subseteq$ `T_p`.

La vérification de la compatibilité n'est pas la seule question qui doit être traitée dans le cadre du niveau syntaxique de l'interopérabilité entre les composants. En se basant sur la description d'interfaces de deux composants, l'autre question est de savoir si l'un des deux composants peut remplacer l'autre dans un système. Cette notion est appelée la substitution et elle est d'une importance capitale dans le développement à base de composants, car elle assure une évolution cohérente du système ainsi que sa maintenance [ZW95]. Au niveau signature, la substitution d'un composant C par un composant C' est traduite par la vérification si (i) toutes les interfaces offertes par C sont aussi offertes par C', (ii) toutes les interfaces requises par C' sont aussi requises par C (C' offre plus de services que C' et demande moins), et (iii) pour chaque opération dans chaque interface partagée, les types des paramètres d'entrée dans C' sont des sous-types des paramètres d'entrée dans C et inversement pour les paramètres de retour. Dans le domaine de la programmation orientée objet, ce principe de sous-typage pour la substitution a été introduit par Barbara Liskov et Jeannette M. Wing sous le nom du *principe de substitution de Liskov* [LW94, LW01].

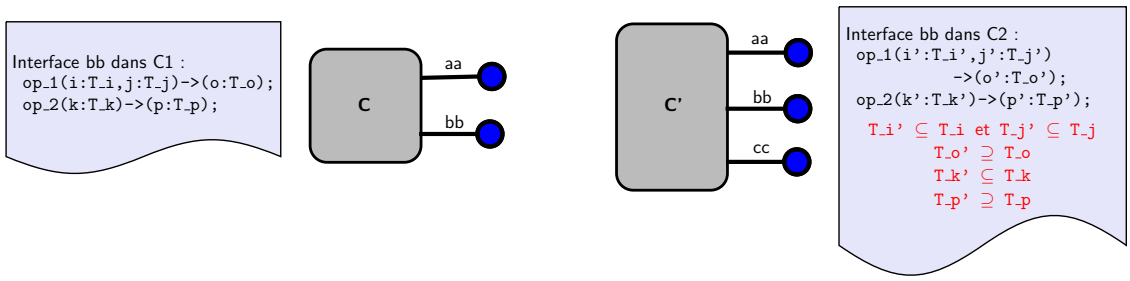


FIGURE 2.5 – Substitution des composants au niveau signature

Comme le montre la figure 2.5, si la règle de sous-typage est valide pour les opérations de l'interface `bb` sont valides aussi pour les opérations de la même interface `bb` dans C et C', alors le composant C peut être substitué par C'.

Malheureusement, les informations fournies au niveau signature ne sont pas suffisantes pour assurer une réutilisation efficace et correcte des composants. Si on considère que l'interopérabilité est basée tout simplement sur les conditions de compatibilité entre les signatures des opérations, d'autres problèmes peuvent avoir lieu, par exemple, l'addition et la soustraction ont les mêmes signatures, mais des comportements complètement opposés. Ce problème peut survenir fréquemment dans le développement des composants logiciels. Par exemple, la majorité des fonctions mathématiques sont souvent sous la forme (`float`,...,`float`)→`float`. Par conséquent, la vérifi-

1. `T'` $\subseteq$ `T` veut dire que `T'` est un sout-type de `T` (i.e., le domaine des valeurs de `T'` est inclus dans celui de `T'`).

cation de la compatibilité sémantique entre les opérations est de nécessité fondamentale.

2.3.2 Niveau sémantique

Le niveau sémantique de l'interopérabilité assure que l'échange des services entre les composants logiciels est sémantiquement correct. À ce niveau, le demandeur et le fournisseur des services doivent avoir une compréhension commune de la signification des services demandés. La vérification de la compatibilité sémantique des composants est une question difficile à traiter. La notion de l'interopérabilité sémantique a été introduit pour la première fois par Sandra Heiler dans [Hei95].

Dans [LS00], les auteurs présentent un aperçu sur les approches proposées dans le cadre de la spécification de la sémantique des interfaces des composants. Chaque proposition utilise des différentes notations en fonction des problèmes traités et les propriétés à vérifier : pré et post-conditions, la logique temporelle, l'algèbre de processus, etc. Dans toutes ces propositions, la compatibilité et substitution sont formellement vérifiées entre les composants.

La compatibilité sémantique veut dire que le comportement fourni par un composant est conforme à celui prévu par ses clients. Plus clairement, la compatibilité peut être prouvée en vérifiant que les préconditions de ses méthodes sont satisfaites par le composant client lors de leurs invocations et que leurs post-conditions satisfassent les attentes des clients. Ce principe est la base du l'approche "design par contrat" proposé par Bertrand Meyer [Mey88].

La figure 2.6 montre que la spécification de l'appel client de la méthode `op_2` dans `C1` est compatible sémantiquement avec la spécification de son implémentation fournie par le composant `C2` si est seulement si la précondition `Pre(op_2,C1)` de l'appel dans `C1` satisfait la précondition `Pre(op_2,C2)` de l'implémentation de la méthode dans `C2` et la post-condition `Post(op_2,C2)` satisfait `Post(op_2,C1)`.

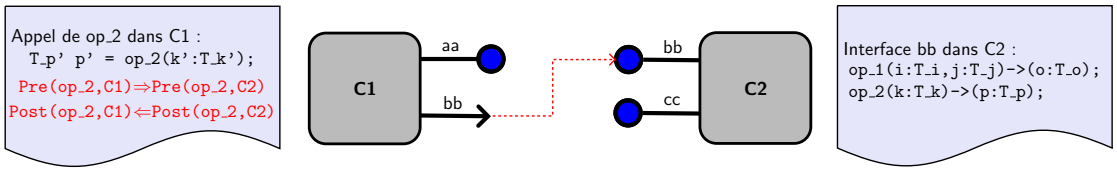


FIGURE 2.6 – Compatibilité des composants au niveau sémantique

De l'autre côté, la substitution dans ce contexte est connue sous le nom "sous-typage comportemental" (*behavioral subtyping*). Cette notion a été introduite pour la première fois, dans le cadre de la programmation orientée objet, par Pierre America dans [Ame91]. Les travaux présentés par America traitent la substitution polymorphique des objets dans les codes clients et la consistance d'une classe fille avec sa classe mère. Le mot comportement désigne la manière selon laquelle les méthodes d'un objet manipulent ses attributs et leurs paramètres. D'autres travaux [LW94, ZW97, DL96] ont suivi les travaux d'America et utilisent des notations différentes pour spécifier le comportement des objets et approfondir les études sur les problèmes du sous-typage comportemental dans le cadre de la programmation orientée objet. Ces travaux utilisent les pré et post-conditions et les invariants pour prouver la correction partielle du sous-typage des objets.

America propose une spécification du comportement des méthodes d'une classe qui implémente un type τ basé sur les pré et post-conditions. Une spécification d'une méthode $m(\bar{p})$ est définie par $\{P\}m(\bar{p})\{Q\}$ tel que $\bar{p}$ est un vecteur qui rassemble les paramètres d'entrée et de

sortie de la méthode, P est la précondition de la méthode, et Q est sa post-condition. L'auteur établit les hypothèses qui doivent être imposées sur deux types τ et σ pour que σ soit un sous-type de τ. Les hypothèses exigent que pour chaque spécification $\{P\}m(\bar{p})\{Q\}$ de chaque méthode $m(\bar{p})$ dans la spécification de τ, il existe une une spécification $\{P'\}m(\bar{p})\{Q'\}$ de $m(\bar{p})$ dans la spécification de σ telle que $P \Rightarrow P'$ et $Q' \Rightarrow Q$. Cette condition garantit que la spécification de la méthode $m(\bar{p})$ dans le sous-type σ de τ est sémantiquement compatible avec la spécification de l'appel client de la méthode si ce dernier est sémantiquement compatible avec la spécification de $m(\bar{p})$ dans τ. Le théorème déduit par l'auteur dit : *si une classe C implémente un type σ et σ est un sous-type de τ, alors C implémente τ et elle peut substituer toute implémentation de τ.* Ce théorème établit le principe de l'héritage objet : si σ est un sous-type de τ alors toutes implémentation de σ hérite de l'implémentation de τ. Pour déduire, une classe C' hérite d'une classe C si est seulement si pour toute méthode $m(\bar{p})$ partagée entre les deux classes la spécification de $m(\bar{p})$ dans C' peut substituer syntaxiquement et sémantiquement sa spécification dans C.

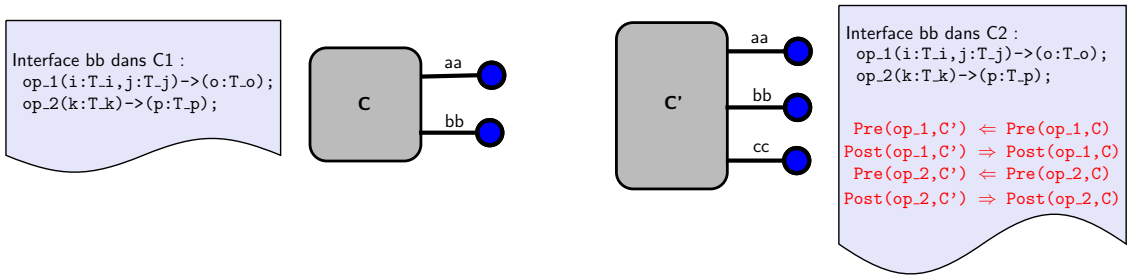


FIGURE 2.7 – Substitution des composants au niveau sémantique

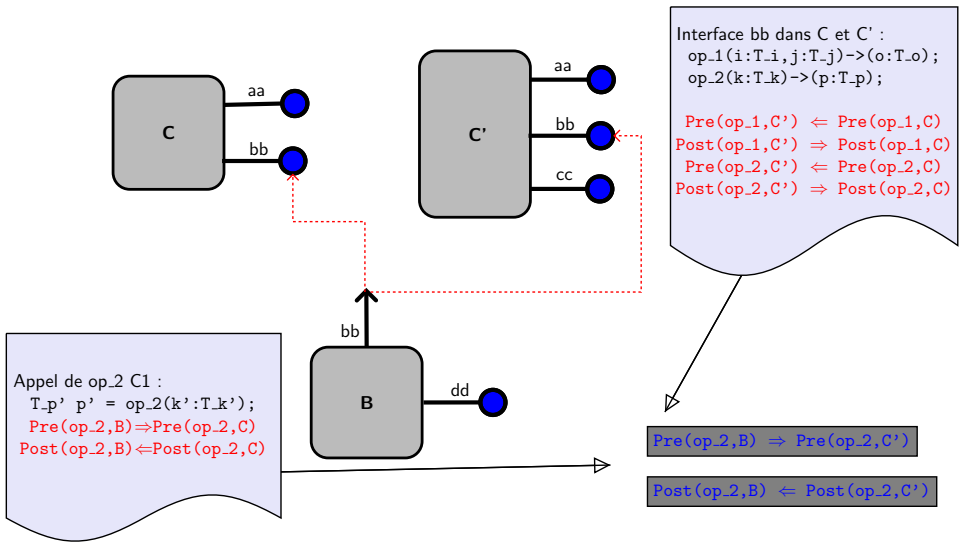


FIGURE 2.8 – Préservation de la compatibilité par la substitution

Les composants peuvent être considérés comme des classes plus évoluées. Par conséquent, les conditions de la substitution sémantique peuvent être appliquées exactement de la même manière que les classes ordinaires. Cependant, nous devons indiquer que la notion de l'héritage

des composants n'est utilisé dans la littérature, on parle plutôt du raffinement. La notion de l'héritage est spécifique aux classes objet et elle est définie rigoureusement vue la standardisation de la notion des classes contrairement aux composants. De manière générale, un composant C' raffine un composant C si toutes les conditions citées dans la section précédente sont satisfaites et pour chaque méthode $m(\bar{p})$ partagée entre les deux, la précondition de $m(\bar{p})$ dans C satisfait la précondition dans C' et vice versa pour les post-conditions. La figure 2.7 illustre les implications entre les pré et post-conditions des méthodes partagées entre un composant C et sa version raffinée C'. La figure 2.8 montre la préservation de la compatibilité sémantique par le sous-typage sémantique ou comportemental.

Le sous-typage comportemental est plus expressif et plus fiable pour vérifier la sûreté de la compatibilité et le raffinement des composants par rapport au sous-typage syntaxique, mais le problème est que le sous-typage syntaxique est décidable et peut être vérifié en un temps raisonnable par contre le sous-typage comportemental est en général indécidable. Toutes ces approches proposées restreignent toujours l'applicabilité de leurs solutions. Par exemple, les pré et post-conditions couvrent uniquement la correction partielle en supposant que les les opérations terminent sans erreurs, mais elles ne couvrent pas la correction totale. Un autre inconvénient de ces approches est qu'aucune parmi elles ne prend en compte la sémantique et les contraintes sous lesquelles les appels externes d'un composant à d'autres sont soumis. Dans ce contexte, Martin Büchi et al. [BW97] proposent des composants boîtes grises dans la sémantique interne de leurs codes et l'enchaînement des appels des services est plus au moins révélé (cf. section 1.2). Dans leur approche, la spécification d'un composant est basée sur des extensions des langages de programmation par des spécifications plus riches des pré et post-conditions, les assertions, et les invariants en tenant en compte des échanges externes effectués par les composants avec leurs environnements.

Heiler [Huc], dans ces travaux, dénombre certains problèmes causés par rendre l'information sémantique explicite au niveau de la vérification de l'interopérabilité entre les composants. Encore plus de difficultés apparaissent lorsque les composants sont utilisés dans des environnements non prévus par leurs développeurs d'origine. L'ingénierie des logiciels est pleine d'exemples qui montrent les anomalies et parfois les catastrophes causées par l'incompatibilité sémantique entre les composants dans les systèmes embarqués.

2.3.3 Niveau protocole

Le niveau protocole de l'interopérabilité entre composants logiciels traite l'ordre relatif selon lequel un composant attend ou prévoit l'appel de ses méthodes, ainsi que les règles qui gouvernent ses interactions et les situations de blocage. Ce niveau d'interopérabilité a été introduit par Yellin et Strom dans [YS97] (cf. section 1.5). Dans le premier chapitre, nous avons présenté les formalismes communément utilisés pour la spécification des protocoles des composants. Dans ce qui suit, nous allons présenter les fondements sur lesquels l'interopérabilité au niveau protocole est basée. Nous détaillons, pareillement comme pour les deux niveaux précédents, les notions de la compatibilité et la substitution des protocoles des composants.

À ce niveau, deux composants sont dits compatibles au niveau protocole si les restrictions imposées sur leurs protocoles comportementaux peuvent être synchronisés sans provoquer des situations de blocage. La vérification de la substitution est aussi possible à ce niveau. Afin de vérifier si un composant C peut être remplacé par un composant C' nous devons vérifier deux points. Premièrement, tous les services fournis par C sont aussi fournis par C' et que les messages sortants de C' (les réponses et les opérations requises) sont un sous ensemble des messages sortants

de C. Deuxièmement, les deux protocoles ne se contredisent pas et donc la relation entre eux doit être soumis à une simulation des traces [YS97].

En partant du critère polyvalent des langages utilisés pour la spécification des protocoles des composants et la difficulté de trouver une description générique de l'interopérabilité à ce niveau, nous avons choisi de présenter deux approches très souvent utilisées. La première est basée sur l'algèbre de processus π-calcul. La deuxième est basée sur les automates. Nous présentons aussi un aperçu sur les autres approches utilisées pour la spécification et la vérification de l'interopérabilité au niveau protocole.

Extension des interfaces CORBA par des protocoles π-calcul

Nous reprenons l'approche présentée dans [CFTV00] introduite dans la section 1.5.2.2. Comme nous l'avons déjà évoqué, l'approche étend les descriptions CORBA IDL des interfaces de composant par des processus π-calcul pour spécifier les protocoles comportementaux. Dans cette section, nous détaillons comment la compatibilité et la substitution des protocoles sont effectuées dans cette approche.

Nous considérons la spécification de l'interface du composant Compte qui gère le compte bancaire d'un client chez une banque. La spécification π-calcul du protocole de ce composant est donnée dans la section 1.5.2.2. Le protocole du client peut être décrit de la manière suivante :

```
protocol Client {
  Client(ref,compte) =
   (^mnt)
     (compte!debiter(mnt,crdInsuff).
       (compte?(recu).zero +
        crdInsuff?(credit,mnt)).zero)
};
```

Le protocole du client permet au client uniquement de débiter le montant `mnt` de son compte. La vérification de la compatibilité entre les deux protocoles `Compte` et `Client` est vérifiée en testant si la composition parallèle de leurs processus π-calcul ne provoque pas une situation de blocage. Pour cela, il faut vérifier si les dérivations du processus π-calcul suivant mènent à l'inaction `zero` :

$$\texttt{(\^{}clt,cmpt,crd) (Client(clt,cmpt) | Compte(cmpt,crd))} \stackrel{\tau}{\Longrightarrow} \texttt{zero}$$

La vérification de la substitution des protocoles est aussi possible dans le cadre de cette approche. La substitution des protocoles est basée sur une version moins restrictive de la bisimulation utilisée pour le raffinement des processus π-calcul. Le lecteur peut consulter [CPT99] pour avoir plus de détail sur la relation de bisimulation utilisée. La procédure d'analyse de l'interopérabilité des processus π-calcul est *NP-difficile*.

Automates CoIN

La composition des automates CoIN (cf. section 1.5.2.4) est basée sur une approche de synchronisation dite "d'attente" (cf. figure 2.9(a)) : si un composant est prêt à effectuer une action d'entrée (respectivement de sortie) il doit attendre (reste bloqué) jusqu'à ce que l'environnement fournit l'action de sortie. Le formalisme des automates d'interfaces et ses dérivés, SOFA [PV02], et d'autres sont basés sur une approche de synchronisation (cf. figure 2.9(b)) qui autorise l'exécution des actions de sortie directement sans attendre que leurs contreparties d'entrée soient

prêtes. Cette approche est plus adaptable aux systèmes à base de composants : l'action d'entrée reste bloquée jusqu'à que son équivalent en sortie est activée par l'environnement.

La substitution d'un automate CoIN par un autre dans une architecture est basée sur une relation d'équivalence en fonction de l'ensemble de labels [2] observables prédéfinis [vVZ07]. Deux automates CoIN A et A' sont reliés par une relation d'équivalence, noté $\equiv_X$, en fonction d'une ensemble d'actions observables $X \subseteq \mathcal{L}_A \cup \mathcal{L}_{A'}$ et qui représente une bisimulation faible entre les états des deux automates C et C'. Les auteurs prouvent que cette relation d'équivalence est préservée entre le produit de $A \otimes E$ et le produit $A' \otimes E$ tels que E est un troisième automate CoIN. Ce résultat représente la validité de la substitution de A par A'. La figure 2.10 représente la relation d'équivalence entre A et A' en fonction d'un ensemble de labels observables X.

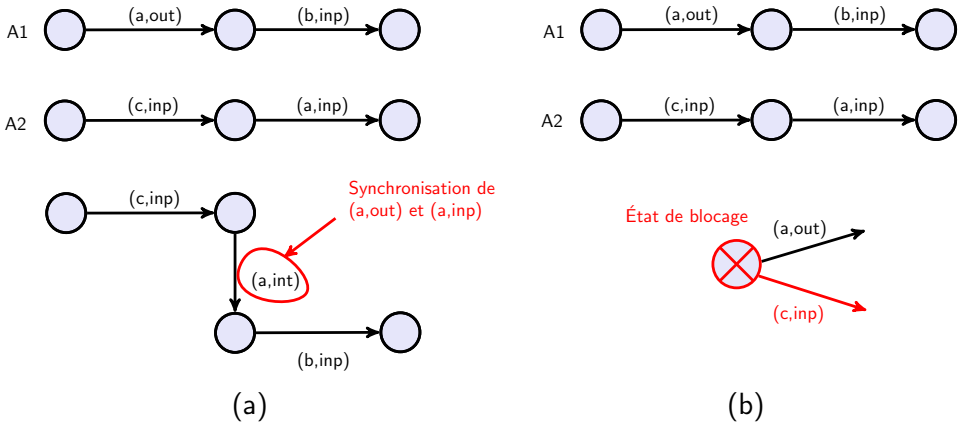


FIGURE 2.9 – Approches de synchronisation des actions partagées

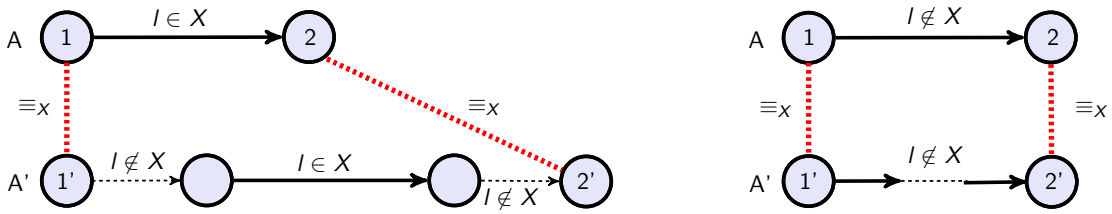


FIGURE 2.10 – Bisimulation faible entre A et A'

Autres approches

Il y a plusieurs autres propositions qui traitent la spécification et l'interopérabilité des composants au niveau protocole. Par exemple, Doug Lea et al. [LM95] ont proposé une extension des interfaces CORBA IDL appelé PSL qui décrit les protocoles des composants des systèmes ouverts. Cette approche est basée sur des règles logiques et temporelles qui relient des situations (états). Les situations PSL sont représentées par des expressions paramétrées qui décrivent les points communs des mondes transitoires des états par lesquels un composant peut passer en

2. L'ensemble de labels $\mathcal{L}_A$ d'un automate CoIN A est défini par l'ensemble $((S_H \cup \{-\}), Act, (S_H \cup \{-\}))$ tel que Act est l'ensemble des actions (d'entrée, de sortie, et internes), S_H représente l'ensemble des identificateurs des composants primitifs qui construisent une hiérarchie de composants H.

fonction de la description de son rôle. Les règles des protocoles PSL décrivent les conditions sous lesquelles les réalisations des situations se produisent. Ces règles sont des collections des situations liées par des opérateurs temporels qui permettent de définir leur ordonnancement. Le test de la compatibilité entre les rôles PSL est traduit par la vérification des propriétés de sûreté et de vivacité des systèmes composites.

Le raffinement des protocoles est traduit par l'introduction de nouvelles règles raffinées applicables dans des situations plus spécifiques que les règles générales sans rendre invalide ces règles générales. Le raffinement PSL peut être catégorisé en deux types : raffinement basé sur les versions et les spécialisations. Le premier type permet d'étendre les règles des protocoles dans un contexte existant en ajoutant des spécificités plus fortes durant le processus de développement. Les spécialisations permettent l'ajout de nouvelles sous-interfaces reliées aux interfaces originales. PSL supporte ce genre de raffinement par des tactiques de sous-typage.

L'approche proposée par Il-Hyung Cho et al. [CMK98] unifie la spécification des protocoles et la sémantique des méthodes pour vérifier l'interopérabilité des composants. Pareillement, les travaux de Jun Han [Han99] étendent les IDLs des composants par des informations sémantiques en termes de contraintes (décrites par un sous ensemble de la logique temporelle) et des rôles d'interaction entre les composants. Par contre, leur approches n'est pas appliquée sur des plate-formes réelles comme CORBA ou EJB et elle n'est pas outillé par des prototypes standards.

Les travaux de Rémi Bastide et al. [BSP99] utilisent les réseaux de Petri pour décrire le comportement des composants CORBA. L'approche mélange aussi la spécification des protocoles et la sémantique des méthodes sans une séparation claire entre les deux niveaux. En outre, l'approche est soumise à certaines limitations imposées par la notation des réseaux de Petri comme le manque de modularité et l'évolutivité des spécifications. Dans la section 1.5.2, d'autres travaux sur les protocoles de composant ont été cités.

L'approche de composition des automates d'interface et automates d'interface sociables (cf. section 1.5.2.4) est considérée comme une approche optimiste [dAH01, ASF+05]. La possibilité de faire interagir les deux automates d'interface dans un environnement qui évite d'atteindre des situations de blocage implique la compatibilité de deux automates d'interface. L'approche de raffinement pour la substitution de ces automates est basée sur une relation de simulation alternée [AHKV98]. La compatibilité et le raffinement des automates d'interface sont détaillés dans le chapitre 3.

2.4 Vérification formelle des propriétés des systèmes à base de composants

Dans les systèmes à base de composants, les méthodes formelles ont commencé à prendre place récemment. Plusieurs approches ont été proposées dans le cadre de la spécification et la vérification formelle de l'assemblage de composants et la correction des systèmes à base de composants. Dans ce contexte, la vérification de l'interopérabilité entre composants peut être étendue par la vérification de la sûreté du système global. La sûreté est établie en vérifiant les relations de conformité entre les propriétés des composants composites et celles de ses sous-composants [Kof07]. Le architectes et les concepteurs doivent spécifier clairement les composants et leurs propriétés par une méthode formelle déductive basée sur les preuves ou un modèle de spécification abstrait basée uniquement sur les spécifications contractuelles des interfaces des composants et leurs protocoles comportementaux (les composants sont des boîtes noires dont les implémentations sont souvent cachées). Les outils basés sur les preuves formelles sont très complexes à maîtriser. C'est pour cela la majorité des approches utilisent la catégorie des

méthodes formelles basée sur les modèles pour spécifier et vérifier les logiciels orientés composant.

Le comportement abstrait évolutif d'un système à base de composants est décrit par ses actions et réactions déduites à partir de celles de ses sous composants. Il est important d'avoir, à un moment donnée, une description claire et explicite des états du système dans le temps. Le modèle abstrait est un modèle fini permettant de décrire les états descriptifs le comportement des composants. Ce modèle peut être décrit par plusieurs formalismes comme les structure de Kripke, les systèmes de transitions étiquetés, les automates de Büchi [Bü66], etc. Le *model-checking* est une technique permettant de décider si ce modèle est conforme à un ensemble de propriétés ou non. Généralement, le *model-checking* est effectué par la vérification si un ensemble de propriétés sont satisfaites par le modèle. Ces propriétés sont exprimées soit par la logique temporelle et vérifiées sur tous les états du système, soit par le même modèle utilisé pour décrire le système et, dans ce cas, elle sont vérifiées par une analyse de conformité avec le modèle de base. Le problème principal de la vérification des propriétés de sûreté est l'explosion combinatoire du nombre d'états du modèle et prennent beaucoup de temps pour être explorés.

2.4.1 Principales approches

Plusieurs travaux ont été publiés dans le contexte de la vérification des l'analyse de sûreté des systèmes à base de composants, nous allons présenter les plus connus et les plus proches de nos travaux.

Darwin & Tracta

Comme nous avons déjà évoqué précédemment dans la section 1.6.2, Darwin [MDEK95] est un langage de spécification des systèmes hiérarchiques à base de composants. Il décrit les structures structures dynamiques des composants évoluant durant l'exécution. Tracta [GKC99b, GKC99a, MKG99] décrit le comportement des composants Darwin. Ce langage est fondé sur les systèmes de transition étiquetés (LTSs). Tracta supporte la vérification des propriétés de sûreté et de vivacité. Les propriétés de sûreté sont exprimées par des LTSs déterministes pour décrire le système désiré. Un système de composants S satisfait une propriété P si et seulement si :

$$\mathit{traces}(S) \setminus \alpha P \subseteq \mathit{traces}(P)$$

où αP représente l'alphabet de la représentation LTS de la propriété P. De manière informelle, le comportement (toutes les traces) du système restreint aux transitions étiquetées par l'alphabet de P doit être inclus dans P.

Les propriétés de vivacité sont spécifiées par des automates de Büchi [Bü66] particuliers. Les automates de Büchi utilisés sont (i) déterministes, (ii) complets (à partir de chaque état d'un automate B, il existe une transition étiquetées par chaque $a \in \alpha B$), et les choix de transitions dans le système S sont supposés être équitables (infiniment souvent activables). Pareillement, le système S satisfait une propriété modélisée par un automate B si B accepte toutes les exécutions ou les traces infinies de S Pour vérifier les propriétés, Tracta utilise le langage LTSA (*Labeled Transition Systems Analyser*) [MK99].

CoIn Tool & DiVinE

L'environnement de vérification CoIN [BBČ+08] est un environnement de vérification des systèmes à base de composants spécifiés par les automates CoIN présentés dans les sections 1.5.2.4 et 2.3.3. L'outil permet de vérifier les interactions entre composants dans les systèmes hiérarchiques

en utilisant des techniques de model-checking basées sur la logique temporelle linéaire LTL (Linear Temporal Logic) [Pri67, MP90, Pnu97]. L'outil utilise une version étendu de la logique LTL, nommée CI-LTL, pour décrire les propriétés des interactions possibles entre les composants en se basant sur le modèles des automates CoIN. La logique est basée uniquement sur les actions. CI-LTL utilise tous les opérateurs LTL standards, comme l'opérateur du prochain état *next* $\mathcal{X}$, l'opérateur *until* $\mathcal{U}$, les opérateurs booléens, etc. Par contre, CI-LTL n'utilise pas des propositions atomiques sur des variables comme opérandes de ses opérateurs. CI-LTL utilise deux sortes de formules $E(l)$ et $P(l)$ telle que l est une étiquette dans l'ensemble $\mathcal{L}_A$ d'un automate CoIN A. Une étiquette représente une interaction entre deux composants via une action partagée. La sémantique des deux formules est définie comme suit :

- $E(l)$ signifie que "l'étiquette l est *activable*" et évaluée vraie si l'interaction représentée par l peut éventuellement se produire.
- $P(l)$ signifie que "l'étiquette l est en cours d'exécution (*the label is proceeding*)" et évaluée vraie si l'interaction représentée par l est entrain de se produire.

La syntaxe de la logique CI-LTL est décrite comme suit :

1. $E(l)$ et $P(l)$ sont des formules telle que l est une étiquette ;
2. si Φ et Ψ sont des formules alors $\Phi \wedge \Psi$, $\neg\Phi$, $\mathcal{X}\ \Phi$, et $\Phi\ \mathcal{U}\ \Psi$ sont aussi des formules ;
3. Toutes les formules peuvent être obtenues par des applications finies de (1) et (2).

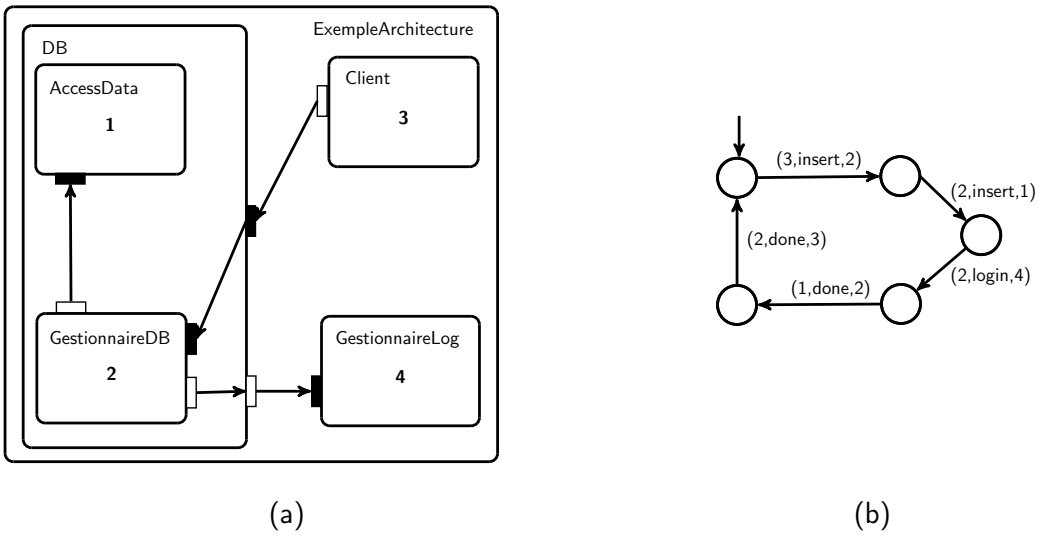


FIGURE 2.11 – Automate CoIN d'une application d'accès à une base de donnée

D'autres opérateurs peuvent être définis comme des cas particuliers des opérateurs de base : $\Phi \vee \Psi = \neg(\neg\Phi \wedge \neg\Psi)$, $\Phi \Rightarrow \Psi = \neg(\neg\Phi \vee \Psi)$, $\mathsf{F}\ \Phi = \mathsf{vrai}\ \mathcal{U}\ \Phi$, et $\mathsf{G}\ \Phi = \neg\mathsf{F}\ (\neg\Phi)$. L'opérateur F (*eventually*) exprime qu'une formule est satisfaite dans le futur et l'opérateur G (*Globally*) exprime qu'une formule est toujours satisfaite. Les propriétés CI-LTL sont évaluées sur les exécutions des automates CoIN. Une exécution $\sigma = s_0, l_0, s_1, l_1, s_2, l_2, ..$ d'un automate CoIN $A_C = (S,Act,\delta,I,H)$ [3] d'un composant C est une séquence infinie telle que s_i sont des états et pour tout i, $(s_i, l_i, l_{i+1}) \in \delta$.

3. S est l'ensemble des états, Act est l'ensemble des actions, δ est l'ensemble des transitions, I est l'ensemble des état initiaux, et H est l'hiérarchie des composants primitifs utilisé dans le composant C.

L'exemple présenté dans la figure 2.11(a) représente une architecture d'une simple application qui gère l'accès à une base de donnée. L'architecture contient quatre composants primitifs AccessData, GestionnaireDB, Client, et GestionnaireLog identifiés respectivement par les identificateurs 1, 2, 3, et 4. Le Client envoie une requête d'insertion d'une ligne dans une table de la base de données grâce à l'action *insert*. Le composant GestionnaireDB demande en conséquence à l'utilisateur de s'identifier en invoquant (action *login*) le composant GestionnaireLog. Après l'insertion des donnée, le composant GestionnaireDB renvoie un acquittement au client en envoyant le message *done*. L'automate CoIN présenté dans la figure 2.11(b) traduit ce processus d'interaction entre les composants. Une étiquette (i_1,a,i_2) représente la synchronisation de l'action a activable en entrée dans i_2 et en sortie dans i_1 pour $i_1, i_2 \in \{1,2,3,4\}$. La propriété temporelle $\mathbf{G}\ (P((3,insert,2)) \Rightarrow \mathbf{F}\ (P((2,done,3))))$ est satisfaite sur toutes les exécutions possibles de l'automate par contre la propriété $\mathbf{G}\ (P((3,insert,2)))$ n'est pas satisfaite.

D'autres travaux de recherche ont étendu le langage DiVinE (*Distributed Verification Environment*) [BBv+06, BBPR08] pour vérifier les automates CoIN. DiVinE est un outil d'analyse d'accessibilité et de vérification de modèle basé sur la logique temporelle linéaire LTL. À partir de la version 2.5, l'outil supporte un nouveau langage d'entrée permettant de spécifier les automates CoIN et les propriétés CI-LTL.

UML, cTLA & SPIN

Dans [KKS10], les auteurs proposent une approche de développement orientée protocole de composants basée sur UML. Les protocoles sont décrits par des diagrammes d'activité UML et leurs sémantiques formelles sont décrites par un langage basé sur la logique temporelle des actions (*Temporal Logic of Actions* - TLA) [Lam02] appelée en anglais *compositional Temporal Logic of Actions* (cTLA) [HK00]. La traduction vers des modèles formels PROMELA [Hol97] est possible ce qui permet à des outils de vérification de modèle comme SPIN de vérifier les propriétés. L'approche tire profit de la standardisation UML (outils graphiques et diagrammes) et la sémantique formelle du langage de spécification cTLA pour vérifier la correction des systèmes orientés composants.

La composition de composants est traduite par la synchronisation des diagrammes de séquence UML des composants avec des diagrammes d'activité selon des règles de synchronisation prédéfinies. La sémantique des diagrammes d'activités résultants est décrite par des processus cTLA. Ces processus sont transformés vers un modèle PROMELA. Le *model-checker* SPIN vérifie si le modèle satisfait l'ensemble de propriétés de sûreté et de vivacité du système décrites dans la logique temporelle linéaire LTL.

Modules d'interface sociables & TICC

Luca de Alfaro et al. [ASF+05] ont proposé un formalisme étendu des automates d'interfaces sociable (cf. section 1.5.2.4) basé sur l'utilisation des variables partagées appelé modules d'interfaces sociables (*Sociable Interface Modules*). La synchronisation entre les actions est basée sur deux principes : (i) le premier est que la même action peut être activable en entrée et en sortie dans automate, et (ii) le deuxième est l'utilisation des variables partagées manipulables par plusieurs modules en même temps. Les auteurs ont proposé également l'outil TICC (*Tool for Interface Compatibility and Composition*) [AAS+06] de vérification de compatibilité et de substitution entre modules basé sur des algorithmes symboliques agissant sur des graphes MDDs [BFG+93, SKMB90]. Les MDDs sont des graphes permettant de représenter et manipuler des fonctions de type $P \rightarrow \{vrai, faux\}$ avec P est l'ensemble des prédicats définis sur un

ensemble de variables V.

TICC permet de vérifier aussi les propriétés de sûreté des modules. Un module d'interface sociable M est sûr ou bien formé si et seulement si toutes ses exécutions respectent un invariant défini par un sous ensemble de l'espace d'états total défini par toutes les valuations possibles des variables manipulables par M.

Dans la section suivante, nous présentons brièvement les techniques d'adaptation des composants logiciels qui ont pour objectif de proposer des solutions aux problèmes où l'interopérabilité des composants n'est pas garantie en utilisant uniquement leurs spécifications contractuelles d'interfaces.

EJBs & Ingénierie dirigée par les modèles

Dans [WSG08], les auteurs proposent des contributions dans le cadre de l'ingénierie dirigée par des modèles (*Model-driven Engineering* MDE) pour la conception des système autonomiques à base de composants EJB. Cette approche est une approche plus pragmatique et donc moins formelle que les approches précédentes. Elle permet de réduire la complexité de développement des applications autonomiques.

Une première contribution consiste à décrire la structure et les fonctionnalités d'un outil MDE qui capture formellement le design de l'application EJB, ses exigences de qualité de service (QoS) et les propriétés autonomiques liées aux EJBs. Cette contribution assure le développement rapide des applications EJB autonomiques par des techniques de génération de code et elle permet de vérifier automatiquement la correction de l'application, et visualiser et interpréter ses exigences QoS complexes.

Une deuxième contribution du travail consiste à décrire comment l'outil MDE génère le code des EJBs pour les installer dans le *framework* de composants. Ce *framework* fournit une structure autonomique pour surveiller, configurer, exécuter les EJBs et lancer leurs stratégies d'adaptation durant le temps d'exécution.

2.5 Adaptation des composants

Le développement à base de composants logiciels a pour but de créer un marché de composants et services réutilisables dans des plateformes hétérogènes, similaire aux marchés des composants matériels. L'un des inconvénients majeurs du développement orienté composant est que, contrairement aux composants matériels, les composants logiciels sont rarement réutilisés tels qu'ils sont [Nie93]. Un certain degré d'adaptation est souvent nécessaire pour assurer l'assemblage et le déploiement des composants. L'adaptation est due aux incompatibilités qui peuvent avoir lieu entre les composants aux différents niveaux d'interopérabilité introduits dans la section 2.3. En effet, ils existent des contextes très spécifiques où les fonctionnalités et les exigences de communication entre les différentes parties d'un système sont clairement établies (comme les librairies mathématiques). Pour le reste des applications, nous ne pouvons pas prévoir qu'un composant fourni par le marché ouvert COTS (*Commercially available Off-The-Shelf*[4]) correspondrait parfaitement aux besoins d'un certain système. Par conséquent, il y aura toujours des incompatibilités qui surviennent durant le processus d'assemblage.

Les composants communiquent à travers leurs interfaces. La réutilisation des composants est donc basée aussi sur leurs interfaces. Les interfaces des composants utilisés dans les modèles

4. Un composant pris sur étagère (*Commercially available Off-The-Shelf* ou COTS) est un mot d'origine anglo-saxonne utilisé pour désigner un composant fabriqué en grande série et non pour un projet en particulier dans le but de réduire les coûts de fabrication et de maintenance.

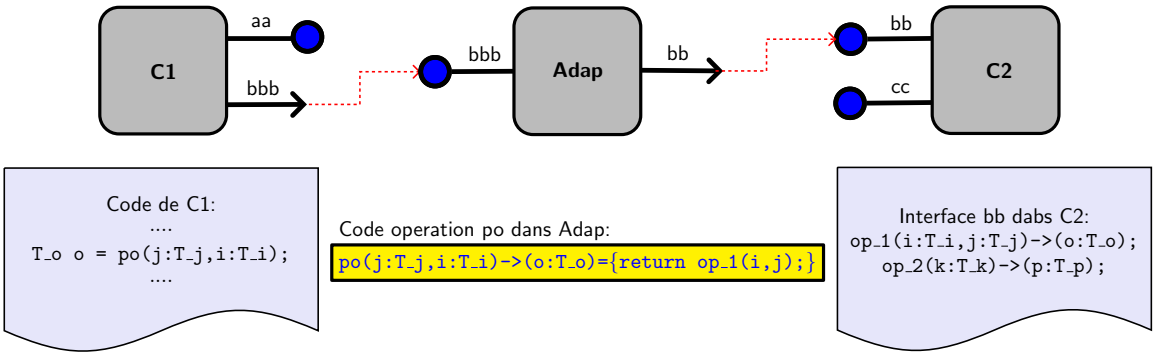


FIGURE 2.12 – Principe d'adaptation

communément utilisés (e.g. EJBs, CCM, .NET) ne permettent pas d'assurer la compatibilité entre eux malgré les disparités possibles entre leurs opérations. Pour résoudre ces problèmes, une nouvelle discipline a vu le jour dans le domaine de développement orienté composants, appelée l'*adaptation logicielle*. L'adaptation est effectuée afin d'éliminer les incompatibilités résultantes entre les composants et assurer une interopérabilité plus flexible entre eux. Elle rend la conception des systèmes beaucoup plus efficace et diminue notamment la tâche du concepteur et du développeur de l'application. L'adaptation logicielle consiste à générer automatiquement des composants médiateurs, appelés *adaptateurs*, pour assurer un assemblage correct de deux ou plusieurs composants malgré les incompatibilités possibles [YS97, CMP08].

Les incompatibilités et les disparités entre les composants aux différents niveaux d'interopérabilité :

- Au niveau signature, les disparités entre les composants sont détectées lorsque les noms des services (offerts/requis) où les messages échangés sont différents, les paramètres des opérations sont organisés selon des ordres différents, leurs types ne sont pas adaptées, ou certains paramètres sont absents ; les composants interagissent souvent par les mêmes signatures des opérations et messages. La figure 2.12 illustre le principe d'adaptation au niveau signature. L'opération `po` invoquée par le composant `C1` correspond à l'opération `op_1` de l'interface `bb` dans le composant `C2` mais elles n'ont pas la même signature. Le composant `Adap` permet de résoudre cette incompatibilité entre `C1` et `C2` en adaptant les signatures des deux opérations ;
- Au niveau protocole, il se peut que les correspondances entre les services requis et offerts ne relient pas exactement un appel avec un seul service. Un appel de service dans un composant n'a pas d'équivalent dans l'autre, ou correspond à plusieurs autres services. De plus, l'ordre des messages ou des actions d'un protocole d'un composant ne recouvre pas celui d'un autre ;
- Au niveau sémantique, le besoin d'adaptation existe aussi, mais il est encore embryonnaire. Par exemple, un composant peut demander un service dont la sémantique est similaire à celle d'un ou plusieurs services offerts par un autre composant. L'adaptation automatique, à ce niveau, est moins évidente. D'autre situations sont possibles, lorsque plusieurs composants offrent des sous services complémentaires pour exécuter la fonctionnalité demandée. Nous précisons qu'une partie de cette thèse traite la problématique de l'adaptabilité sémantique entre les contrats des composants. Le chapitre 5 présente une approche d'adaptation basée sur les automates d'interface enrichis par la sémantique des opérations.

L'adaptation des composants doit respecter un certain nombre de règles. Ces règles ont pour objectif de garantir que les propriétés des composants ne seront pas modifiées après le processus d'adaptation. L'adaptation doit garantir que les clients d'un composant ne doivent pas se rendre compte qu'un composant est adapté ou non. En d'autres termes, les règles d'usage d'un composant adapté par ses clients sont les mêmes que celles de sa version originale. De même, l'adaptation ne doit pas influer sur l'implémentation d'un composant. La version adaptée d'un composant ne doit pas pouvoir être déduite uniquement de ses interfaces mais plutôt en fonction des schémas de communication avec ses clients. Le composant doit d'ailleurs resté utilisable comme une entité autonome. Son implantation reste cachée vis-à-vis de la technique d'adaptation.

2.5.1 Méthodologies d'adaptation

Les méthodologies d'adaptation des composants logiciels sont souvent divisées en deux étapes principales : la spécification de l'adaptateur et sa dérivation. Les descriptions de ces étapes peuvent être illustrées comme suit :

- La spécification de l'adaptateur : les IDLs des composants sont étendues par des règles d'adaptation pour décrire un niveau plus élevé d'interopérabilité et relier les opérations dont les signatures sont différentes et qui sont censées être synchronisées. Les notations utilisées doivent être abstraites et ne se préoccupent pas des spécificités liées au comportement interne de l'adaptateur.
- La dérivation de l'adaptateur : l'adaptateur concret est automatiquement généré en utilisant sa spécification et les interfaces des composants impliqués. La sortie de ce processus est l'adaptateur permettant de faire opérer les deux composants en se basant sur la spécification donnée.

Bien que plusieurs problèmes d'adaptation ont été largement étudiés et traités dans le cadre des différents domaines du développement orienté composant, il a été récemment admis que l'adaptation logicielle doit être considérée comme une thématique à part entière.

2.5.2 Principales approches

Dans la littérature, il est communément admis que l'état de l'art a donné une solution aux questions d'adaptation des composants au niveau signature. Les préoccupations de recherche sont orientées actuellement vers les deux autres niveaux d'interopérabilité : les niveaux protocole et sémantique. Dans cette section, nous présentons brièvement deux approches basées sur l'utilisation de l'algèbre de processus π-calcul et les systèmes de transition étiquetés ainsi que d'autres approches.

Adaptation des protocoles π-calcul

Dans [BBC05], les auteurs ont proposé une approche fondée sur l'algèbre des processus pour générer automatiquement des adaptateurs pour les protocoles des composants inadéquats. L'approche proposée traite le problème de l'adaptation des protocoles incompatibles des composants. La spécification des interfaces des composants est étendue par des descriptions de leurs protocoles comportementaux décrits dans une variante de l'algèbre de processus π-calcul. L'approche est basée sur les étapes de la spécification et la dérivation d'un protocole adaptateur pour deux protocoles donnés. La variante de π-calcul proposée est la suivante :

$$E ::= 0 \mid a.E \mid (x)E \mid [x = y]E \mid E \parallel E \mid E + E$$

$$a ::= \tau \mid x\,?\,(d) \mid x\,!\,(d)$$

Les actions d'entrée et de sortie, qui représentent les appels des opérations et les réceptions de ces appels, sont représentées par $x\,?\,(d)$ et $x\,!\,(d)$ tels que x est le nom de l'action et d est un tuple de paramètres ou de données envoyées ou reçues à travers x. Les actions non observables sont représentées par τ. Les actions sont inclues dans des processus E tels que 0 représente l'inaction, $(x)E$ représente la création d'un nouveau nom x dans un processus E. L'opérateur $[x = y]E$ représente le comportement conditionnel : $[x = y]E$ se traduit par E si $x = y$ et par l'inaction 0 dans le cas contraire. L'opérateur de choix non déterministe ($+$) et l'opérateur ($\parallel$) sont définis par : E + E' est traduit par l'exécution de E ou E', tandis que $E \parallel E'$ se traduit par l'exécution parallèle de E et E'.

role Client = {
  signature request ! (Data *url*) ; reply ? (Data *page*) ;
  behaviour request ! (*url*) . reply ? (*page*).0
}

role Serveur = {
  signature query ? (Data *url*) ; return ! (Data *file*) ;
  behaviour query ? (*url*) . return ! (*file*).0
}

Cet exemple montre les protocoles de deux composants Client et Serveur. Le Client est un navigateur web dans lequel l'utilisateur saisi une URL et envoie la requête pour visualiser la page web désirée ou ouvrir un fichier. Le Serveur est un hébergeur de sites qui contient une application qui permet de recevoir les requêtes des clients et envoyer soit une page web ou un fichier. La spécification du comportement est définie dans la section **behaviour**. Les protocoles de Client et Serveur sont représentés par des processus π-calcul et traduisent le rôle de chacun comme nous venons de le détailler.

Les connexions entre *request* ! et *query* ? et entre *reply* ? et *return* ! représentent la fonction d'adaptation ou la spécification minimale de l'adaptateur entre Client et Serveur. Cette fonction est définie comme suit :

$$M = \{request\,!\,(url) <> query\,?\,(url); reply\,?\,(page) <> return\,!\,(file)\}$$

Intuitivement, la fonction d'adaptation M représente la spécification minimale de l'adaptateur qui vas jouer le rôle du composant au milieu entre les deux composants Client et Serveur. Le comportement d'un adaptateur A qui satisfait la spécification M peut être décrit de la manière suivante :

$$A = request\,?\,(url)\ .\ query\,!\,(url)\ .\ return\,?\,(file)\ .\ reply\,!\,(page)\ .\ 0$$

Le processus de génération du protocole adaptateur de deux protocoles P et Q de deux composants permet de dériver automatiquement à partir de P, Q et la fonction d'adaptation un protocole A qui satisfait les deux propriétés suivantes :

1. la composition parallèle $P \parallel A \parallel Q$ ne provoque pas des situations de blocage ;

2. *A* satisfait toutes les correspondances entre les actions et les dépendances entre leurs paramètres.

Adaptation basée sur les systèmes de transitions étiquetées

Les travaux présentés dans [CPS06, CPS08] sont basés sur l'utilisation des expressions régulières et les systèmes de transitions pour la spécification des protocoles. L'approche proposée, comme la précédente, traite les incompatibilité au niveau protocole des composants. Les auteurs spécifient l'adaptateur par des règles d'adaptation modélisées par des vecteur de synchronisation permettant de décrire les relations entre les actions incompatibles au niveau signature et qui sont censées être synchronisées. Les protocoles de composants sont spécifiés par des systèmes de transitions étiquetés LTSs.

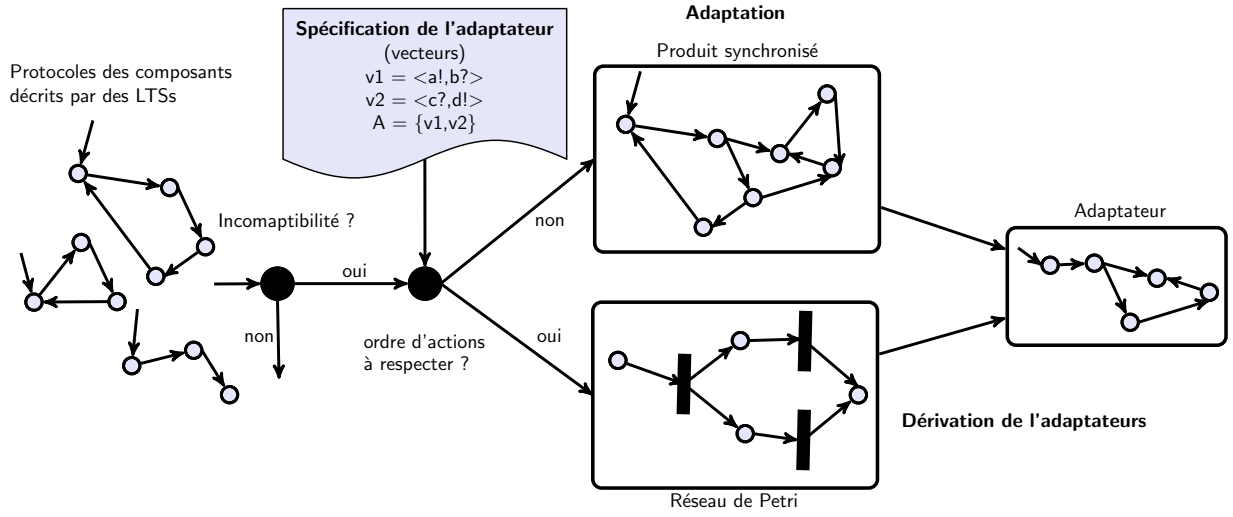


FIGURE 2.13 – Approche d'adaptation présentée dans [CPS06]

La génération automatique de l'adaptateur est basée sur deux approches d'adaptation : l'approche sans ordre et celle en respectant l'ordre des actions dans la spécification. La première approche permet de créer un adaptateur qui ignore l'ordre selon lequel les actions sont organisées dans les systèmes de transitions des composants communicants. L'adaptateur est tout simplement représenté par le produit synchronisé des LTSs de l'ensemble de composants à adapter en inversant les actions d'entrée (resp. de sortie) présentes dans les règles d'adaptation par des actions de sortie (resp. d'entrée). La deuxième est basée sur le respect de l'ordre des actions défini par les spécifications LTS des protocoles des composants. Les auteurs utilisent les réseaux de Petri pour refléter l'ordre des actions dans l'adaptateur. La figure 2.13 illustre le processus suivi pour la génération automatique d'un adaptateur pour plusieurs composants en se basant sur sa spécification et leurs représentations LTSs.

Autres approches

Bosch [Bos00] donne un large aperçu sur les mécanismes d'adaptation, y compris celles non automatisées. Dans [VW01], les auteurs ont développé l'outil PaCoSuite pour modifier visuellement les composants, et de générer des adaptateurs en utilisant les signatures des méthodes et des protocoles des interfaces. Dans [PdAHSV02], les auteurs utilisent la théorie des jeux pour

décider si les interfaces des composants incompatibles peuvent être rendues compatibles en utilisant des techniques d'adaptation. Dans [SG03], les auteurs ont montré comment utiliser les transformations de protocoles pour augmenter l'interaction des comportements d'un ensemble de composants. Cette approche porte sur les problème du renforcement des interactions entre les composants. Dans [Hem05], David Hemer a proposé, à l'aide du langage CARE, des stratégies d'adaptation de composants en utilisant la bibliothèque de modèles paramétrés. L'approche fournit plusieurs modèles d'adaptation et étudie l'adaptation et la composition semi automatisées par les modèles définis. Dans [MLS06], les auteurs ont proposé un modèle d'adaptateurs utilisant la méthode B [Abr96], et permettant de définir l'interopérabilité entre les composants. Dans [AEB08, BHB08], les auteurs proposent un *framework* de développement de composants permettant de générer automatiquement des adaptateurs pour les composants incompatibles au temps d'exécution. Les spécifications d'adaptateurs sont réalisées par les machines à états (*State Machine Diagrams*) d'UML 2.

Dans [SEG08], les auteurs proposent une étude de l'art sur les approches d'adaptation semi-automatiques et automatiques des composants et services. Dans [DBMN08], les auteurs discutent la notion de la compatibilité au niveau protocole des services Web et proposent une approche d'adaptation pour les services incompatibles. Leur proposition traite des notions comme la substituabilité et la contrôlabilité. Dans [Pad09], l'auteur propose un langage formel pour décrire les contrats des services Web. Le travail fournit une sémantique pour définir les règles d'adaptation entre les services en terme des relations d'équivalence. Dans [MPM08], les auteurs proposent une approche d'adaptation entre les services Web qui intègre les trois niveaux d'interopérabilité des composants : signature sémantique et protocole. Le niveau sémantique est décrit par une notation SAWSDL [5] complémentée par une notation BPEL.

2.6 Synthèse

Dans ce chapitre, nous avons expliqué l'interopérabilité des composants et comment elle est mise en œuvre en utilisant les spécifications d'interface aux niveaux signature, sémantique, et protocole. Ensuite, nous avons présenté les travaux qui concernent la vérification formelle de la sûreté des systèmes à base de composants avant et après le processus d'assemblage. À la fin du chapitre, nous avons présenté quelques approches sur l'adaptation de composants logiciels.

5. `http://www.w3.org/TR/sawsdl/`

Chapitre 3

Automates d'interface et l'approche optimiste

Dans ce chapitre, nous allons présenter en détail le formalisme des automates d'interface utilisé pour spécifier les protocoles des composants logiciels. Nous détaillons son approche optimiste de composition qui considère que deux composants sont compatibles s'il existe au moins un environnement convenable avec lequel les deux composants peuvent interagir sans provoquer des situations de blocage.

Ce chapitre présente les définitions formelles des concepts liées aux automates d'interface notamment celles qui statuent la compatibilité et la substitution des composants. Les définitions du produit synchrone et la composition des automates d'interface sont ainsi fournies. Dans la section 3.1, nous présentons un aperçu informel sur les caractéristiques du modèle des automates d'interface. Dans la section 3.2, nous présentons les définitions du formalisme. Dans la section 3.3, nous expliquons comment la compatibilité entre les automates d'interface est vérifiée dans le cadre de l'approche optimiste. La section 3.4 présente l'algorithme de la vérification de la compatibilité et du calcul de composition entre les automates d'interface. Dans la section 3.5, nous présentons l'approche de simulation alternée utilisée pour calculer le raffinement des automates d'interface.

3.1 Aperçus informels

Les automates d'interface (IAs) ont été introduits par Luca de Alfaro et Thomas Henzinger [dAH01, AS04, AH05, AH01] pour modéliser les comportements, évolutifs dans le temps, des composants au niveau des interfaces. Un composant logiciel est conçu conformément aux hypothèses sur son environnement. En d'autres termes, la conception d'un composant décrit un comportement conforme à un ensembles d'hypothèses que l'environnement doit satisfaire. Par exemple, la conception d'un composant orientée objet est basée sur le principe que les appels des méthodes doivent être organisés dans un ordre spécifique pour qu'il puisse agir comme prévu. Les automates d'interface satisfont ces critères de communication entre le composant et son environnement. Un automate d'interface donne un ordre temporel sur l'enchaînement des évènements d'entrée et de sortie provoqués par un composant sous l'hypothèse qu'un évènement d'entrée est la cause autorisant le déclenchement d'un évènement de sortie. Ces évènements sont appelés des *actions*.

3.1.1 Catégories des actions

Les actions d'un automate d'interface sont classées en trois catégories : les actions d'entrée, de sortie et internes.

Actions d'entrée

Les actions d'*entrée* modélisent :

- les méthodes implémentées par le composant qui peuvent être appelées par les clients du composant,
- la réception des valeurs de retour des appels de méthodes,
- la détection par le client des exceptions qui peuvent être provoquées après l'appel d'une méthode,
- la réception d'un message à partir d'un canal de communication.

Une action d'entrée peut représenter l'hypothèse qu'une méthode implémentée par le composant peut être acceptée à un moment particulier durant son cycle de vie. Par exemple, l'opération d'authentification offerte par un serveur pour assurer la connexion d'un client ne peut pas être exécutée que si le client n'est pas connecté. Elle peut aussi modéliser le stockage d'une valeur de retour d'une méthode dans une variable après son appel.

Une action d'entrée peut aussi représenter la détection d'une exception après l'appel d'une méthode. Par exemple, en java, la clause `catch` permet de détecter les exceptions d'une suite d'instructions dans une clause `try`. La clause `catch` peut être donc traduite par une action d'entrée. Une action d'entrée peut également spécifier la réception d'un message à partir d'un canal de communication entre deux composants. À titre d'exemple, les puits d'événements asynchrones des composants CORBA (cf. section 1.7.2.2) peuvent être représentés par des actions d'entrée.

Actions de sortie

Les actions de *sortie* modélisent :

- les appels des opérations de l'environnement,
- l'envoi des valeurs de retour des méthodes,
- l'envoie des exceptions qui se produisent pendant l'exécution des méthodes,
- les transmissions des messages dans un canal de communication. Par exemple, les sources d'événements asynchrones des composants CORBA.

Actions internes

Les actions *internes* représentent les opérations locales à l'intérieur du composant. Par exemple, les appels des méthodes statiques peuvent être représentés par des actions internes.

3.1.2 Approche optimiste

Les automates d'interface sont syntaxiquement similaire aux automates I/O [LT87]. Les automates d'interface de deux composants interagissent par la synchronisation des actions d'entrée et les actions de sortie qui représentent les opérations partagées par leurs composants. Contrairement au automates I/O, il n'est pas nécessaire que toutes les actions d'entrée soient activables à partir de tous les états de l'automate. Cette propriété exprime l'assertion qu'à chaque état,

l'environnement peut générer un sous ensemble des actions d'entrée (approche « optimiste » ou « contrainte par l'environnement »). En capturant les contraintes de l'environnement et en libérant les développeurs de l'obligation de fournir les entrées non fournies par l'environnement, les automates d'interface fournissent une notation concise et formelle conforme à la nature de la conception à base de composants [dAH01].

La composition de deux automates d'interface peut contenir des états *illégaux* ou de *blocage*. À partir de ces états, l'un des deux automates sollicite une action d'entrée qui n'est pas acceptée par l'autre. Par exemple, un composant appelle une méthode susceptible de provoquer des exceptions et il ne prévoit pas leurs traitements. En Java, une situation de blocage est traduite par la détection d'une exception après l'appel d'une méthode non inscrite dans une clause `try`/`catch`. L'exception provoquée par l'exécution de la méthode représente l'action de sortie et le non traitement de l'exception par l'appelant de la méthode représente l'absence de l'action d'entrée correspondante.

La présence de ces états n'implique pas l'incompatibilité des deux automates contrairement aux approches existantes considérées pessimistes. Si chaque automate prévoit que l'environnement fournit uniquement les entrées légales, alors leur composition est restreintes aux exécutions permettant d'éviter l'accessibilité aux état illégaux. L'existence d'un environnement légal pour la composition des deux automates indique qu'ils sont mutuellement *compatibles* et les hypothèses sous lesquelles l'environnement est soumis sont satisfaites. Plus clairement, l'existence des états illégaux ne provoque pas forcément l'incompatibilité des deux automates. La possibilité de faire interagir les deux automates d'interface dans un environnement qui sollicite les actions d'entrée appropriées dans leur produit synchronisé en évitant d'atteindre les états illégaux implique la compatibilité de A_1 et A_2. Les deux automates sont dits incompatibles s'il n'y a aucun environnement permettant d'éviter que leur composition atteigne les états illégaux. Au niveau algorithmique, la vérification de la compatibilité de A_1 et A_2 est effectué en suivant les étapes suivantes :

1. calculer le produit synchronisé $A_1 \otimes A_2$ si c'est possible ;
2. calculer l'ensemble des états illégaux dans $A_1 \otimes A_2$;
3. calculer l'ensemble des états incompatibles dans $A_1 \otimes A_2$ à partir des états illégaux : ce sont les états à partir desquels l'environnement ne peut pas éviter d'atteindre les états illégaux en une ou plusieurs étapes.
4. calculer $A_1 \parallel A_2$ en enlevant de $A_1 \otimes A_2$, les états illégaux, les états incompatibles et les états non atteignables à partir de l'état initial.

Si $A_1 \parallel A_2$ n'est pas vide alors A_1 et A_2 sont compatibles. Sinon, ils sont incompatibles.

Un composant peut être raffiné et sa version raffinée le remplace dans l'architecture des composants. Dans les approches traditionnelles (pessimistes), les contraintes de l'environnement ne sont pas prises en compte. L'interface d'un composant est spécifiée en faisant abstraction de l'environnement et ses besoins. Dans l'approche optimiste, l'interface spécifie le comportement légal du composant et en même temps elle respecte les hypothèses sur le comportement admissible de l'environnement. Un composant raffiné respecte sa version d'origine et l'environnement. Plus précisément, le raffinement d'un composant est traduit par étendre son comportement sans restreindre le comportement admissible de l'environnement. Le raffinement agit de manière contravariante sur les hypothèses d'entrée de l'environnement et le comportement de sortie : un composant offre plus de service et en demande moins. Cela conduit à représenter la relation entre la spécification d'un composant et celle de sa version raffinée par une relation de *simulation alternée* [AHKV98].

3.2 Définition formelle

Définition 3.1 (*Automate d'interface*)**.** *Un* automate d'interface A *est représenté par le tuple* $\langle S_A, i_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ *tels que :*

- S_A *est ensemble fini d'états. A est dit "vide" si* $S_A = \emptyset$ *;*
- $i_A \in S_A$ *est l'état initial ;*
- Σ_A^I, Σ_A^O *et* Σ_A^H *représentent, respectivement, les ensembles des actions d'entrée, de sortie et internes. L'ensemble des actions de A est noté par* Σ_A *;*
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *est l'ensemble des transitions entre les états.*

Nous exigeons que les automates d'interface sont déterministes, i.e. si $(s, a, s_1) \in \delta_A$ *et* $(s, a, s_2) \in \delta_A$*, alors* $s_1 = s_2$*.*

L'ensemble Σ_A^{ext} des actions *externes* d'un automates d'interface A est défini par l'ensemble $\Sigma_A^{\text{I}} \cup \Sigma_A^{\text{O}}$. L'ensemble Σ_A^{loc} des actions *localement contrôlées* de A est défini par l'ensemble $\Sigma_A^{\text{O}} \cup \Sigma_A^{\text{H}}$. Nous définissons par $\Sigma_A^{\text{I}}(s)$, $\Sigma_A^{\text{O}}(s)$, $\Sigma_A^{\text{H}}(s)$, $\Sigma_A^{\text{ext}}(s)$, et $\Sigma_A^{\text{loc}}(s)$, les actions d'entrée, de sortie, internes, externes, et localement contrôlées à l'état s. $\Sigma_A(s)$ représente l'ensemble de toutes les actions activables à l'état s. Un automate d'interface A est fermé si $\Sigma_A^{\text{I}} = \Sigma_A^{\text{O}} = \emptyset$. Un automate d'interface fermé ne peut pas communiquer avec l'environnement. L'alphabet d'un automate d'interface est constituée des noms de ses actions annotées par « ? » pour les actions d'entrée, « ! » pour les actions de sortie et par « ; » pour les actions internes.

Les actions d'entrée ne sont pas nécessairement activables à tous les états d'un automates d'interface A c'est à dire $\Sigma_A^{\text{I}}(s) \subseteq \Sigma_A^{\text{I}}$ pour tout $s \in S_A$ (*non-input-enabled approach*). Nous indiquons également que pour tout $s \in S_A$, $\Sigma_A(s) \neq \emptyset$ ou $\Sigma_A(s) = \emptyset$ (les états à partir desquels aucune transition n'est activable sont autorisés dans un automate d'interface).

Exécutions d'un automate d'interface

Une *exécution* σ d'un automate d'interface non vide A est une séquence alternée $s_0 a_0 s_1 a_1 ... s_n$ d'états et d'actions telle que $(s_i, a_i, s_{i+1}) \in \delta_A$ pour tout $0 \leq i < n$. Une exécution est dite *autonome* si $a_i \in \Sigma_A^{\text{O}} \cup \Sigma_A^{\text{H}}$ pour tout $0 \leq i < n$. Une exécution est *invisible* si $a_i \in \Sigma_A^{\text{H}}$ pour tout $0 \leq i < n$. Un état $s' \in S_A$ est atteignable de manière autonome (resp. invisible) à partir d'un état $s \in S_A$ s'il existe une exécution autonome (resp. invisible) dont l'état de départ est s et l'état final est s'. L'état s est *atteignable* dans A ssi s est atteignable à partir de l'état initial i_A. Tous les états d'un automate d'interface sont atteignables par définition. Nous indiquons que les notations utilisées dans ce chapitre restent les mêmes dans le reste des définitions de la thèse.

Exemple 3.1. Nous considérons un composant logiciel qui implémente un simple service de transmission de messages. Le composant *Comp* a une méthode *msg* utilisée pour envoyer un message. Si la méthode est appelée, le composant renvoie au *Client* un message *ok* pour indiquer s'il le message a été envoyé avec succès ou une exception *echec* en cas d'échec. Pour assurer l'envoi, le composant utilise un canal de transmission qui fournit une méthode *envoyer*. Les deux actions *ack* et *nack* indiquent la terminaison de la transmission avec succès ou non. Le composant *Comp* fait une deuxième tentative d'envoi en cas d'échec. Le composant *Comp* est utilisé par un composant *Client* qui s'attend à ce que les messages soient tous envoyés avec succès et il ne traite pas les échecs d'envoi. Les automates d'interface des composants *Client* et *Comp* sont présentés dans la figure 3.1. L'automate d'interface du composant *Client* a trois actions externes (une action de sortie et deux actions d'entrée) et pas d'actions internes [dAH01].

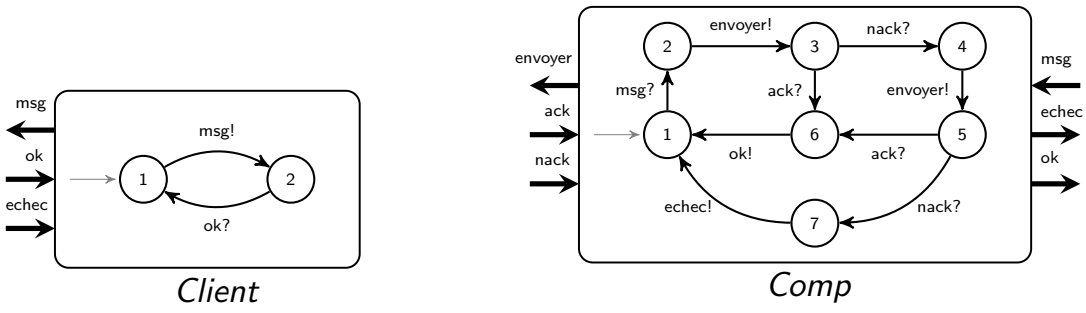


FIGURE 3.1 – Automates d'interface *Client* et *Comp*

L'action *echec* est une action d'entrée dans *Client* mais elle n'est pas activable après l'appel de la méthode *msg*. Ceci veut dire que le *Client* ne traite pas l'exception *echec* susceptible de survenir après l'appel de la méthode *msg*. En important une interface d'un composant, le client connait toutes les signatures de ses méthodes et leurs exceptions. En effet, en invoquant la méthode *msg*, le client est susceptible de savoir qu'elle peut provoquer une exception. Par conséquent, l'action *echec* doit être présente dans l'ensemble des actions d'entrée du composant *Client* malgré l'absence des transitions étiquetées par cette action dans l'automate. ■

3.3 Composabilité, compatibilité et composition

La composition de deux automates d'interface ne peut réussir que si les ensembles de leurs actions d'entrée sont disjoints. De même pour les actions de sortie, l'ensemble des actions internes de l'un des deux automates est disjoint avec l'ensemble des actions de l'autre. Nous mentionnons que la composabilité de deux automates d'interface préserve le déterminisme de leur composition.

Définition 3.2 (*Composabilité*)**.** *Deux automates d'interface A_1 et A_2 sont* composables *ssi* (*si et seulement si*)

$$\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{I}}_{A_2} = \Sigma^{\mathrm{O}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2} = \Sigma^{\mathrm{H}}_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^{\mathrm{H}}_{A_2} = \emptyset.$$

Nous notons par $Partagées(A_1, A_2) = (\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}) \cup (\Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A_1})$ *l'ensemble des actions partagées par A_1 et A_2.*

Nous pouvons définir maintenant le produit entre deux automates d'interface composables. Comme les automates d'interface ne sont pas nécessairement *input-enabled*, alors quelques transitions peuvent être absentes dans le produit.

Définition 3.3 (*Produit synchronisé* $\otimes$)**.** *Soient A_1 et A_2 deux automates d'interface composables, le* produit synchronisé $A_1 \otimes A_2$ *de A_1 et A_2 est défini par*

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ *et* $i_{A_1 \otimes A_2} = (i_{A_1}, i_{A_2})$ *;*
- $\Sigma^{\mathrm{I}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus Partagées(A_1, A_2)$ *;*
- $\Sigma^{\mathrm{O}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus Partagées(A_1, A_2)$ *;*
- $\Sigma^{\mathrm{H}}_{A_1 \otimes A_2} = \Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2} \cup Partagées(A_1, A_2)$ *;*
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *si*

- $a \notin$ *Partagées*$(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$ *ou*
- $a \notin$ *Partagées*$(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$ *ou*
- $a \in$ *Partagées*$(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

L'incompatibilité entre deux automates d'interface A_1 et A_2 pourrait se produire à cause de l'existence des états (s_1, s_2) dans le produit $A_1 \otimes A_2$ tel qu'il existe au moins une action a dans *Partagées*(A_1,A_2) activable à partir de s_1 et elle ne l'est pas à partir de s_2 ou vice versa. Ces états sont dits *illégaux* dans le produit $A_1 \otimes A_2$.

Définition 3.4 (*États illégaux*)**.** *L'ensemble des* états illégaux *Illégaux*$(A_1,A_2) \subseteq S_{A_1} \times S_{A_2}$ *dans* $A_1 \otimes A_2$ *est défini par* $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid (\exists\ a \in$ *Partagées*$(A_1, A_2) \mid$ *la condition suivante est satisfaite* $)\}$

$$\left(\begin{array}{c} (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \notin \Sigma^{\mathrm{I}}_{A_2}(s_2)) \\ \vee \\ (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \notin \Sigma^{\mathrm{I}}_{A_1}(s_1)) \end{array} \right)$$

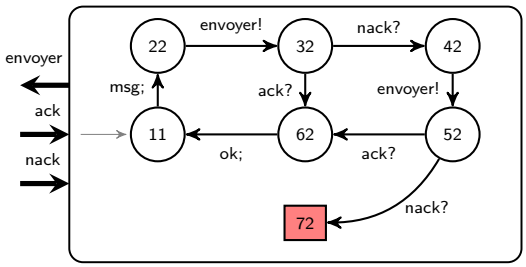


FIGURE 3.2 – Le produit *Comp*⊗*Client*

Exemple 3.2. Le produit des deux composants *Client* et *Comp* est présenté dans la figure 3.2. Le produit présenté ne contient que les état atteignables. L'état (7,2) dans le produit est illégal parce que l'action *echec* $\in$ *Partagées*(*Comp*,*Client*) et la transition (7,*echec*,1) dans *Comp* n'a pas une contre partie dans le composant *Client* à partir de l'état 2. ■

Si le produit $A_1 \otimes A_2$ est fermé (si $\Sigma^{\mathrm{I}}_A = \Sigma^{\mathrm{O}}_A = \emptyset$), A_1 et A_2 sont compatibles s'il n'y a pas d'états illégaux atteignables dans $A_1 \otimes A_2$. Par contre, si $A_1 \otimes A_2$ est ouvert l'accessibilité des états dans *Illégaux*(A_1, A_2) n'implique pas nécessairement que A_1 et A_2 sont incompatibles. L'existence d'un environnement qui pourrait générer les entrées appropriées au produit $A_1 \otimes A_2$ garantit que les états illégaux ne soient pas atteignables. Dans ce cas, les deux automates d'interface A_1 et A_2 peuvent être composés correctement sans donner lieu à des incompatibilités.

Définition 3.5 (*Environnement légal*)**.** *Soient* A_1 *et* A_2 *deux automates d'interface composables, un* environnement légal E *pour* A_1 *et* A_2 *est un environnement* $A_1 \otimes A_2$ *tel que (1)* E *est composable avec* $A_1 \otimes A_2$*, (2)* E *est non vide, (3)* $\Sigma^{\mathrm{I}}_E = \Sigma^{\mathrm{O}}_{A_1 \otimes A_2}$*, (4)* *Illégaux*$(A_1 \otimes A_2, E) = \emptyset$*, et (5) il n'existe pas d'états dans* *Illégaux*$(A_1, A_2) \times S_E$ *atteignables dans* $(A_1 \otimes A_2) \otimes E$.

Exemple 3.3. L'automate d'interface *Canal* de la figure 3.3 est un environnement légal pour le produit de *Client* et *Comp* parce que, dans le produit (*Comp*⊗*Client*)⊗*Canal* de la figure 3.4, l'état (7,2) n'est pas atteignable pour tous les états de l'automate *Canal*. ■

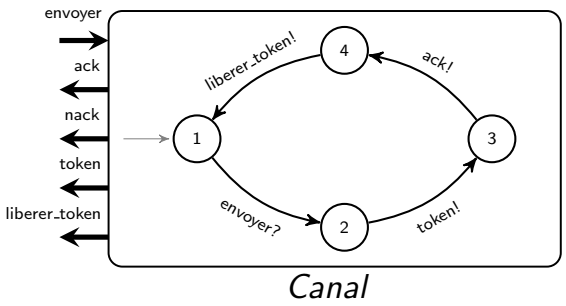


FIGURE 3.3 – Un environnement légal *Canal* pour *Client* et *Comp*

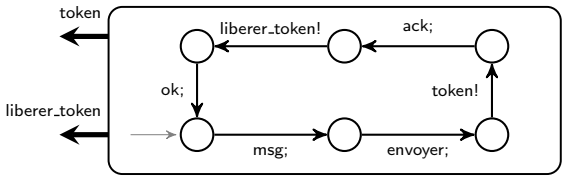


FIGURE 3.4 – Le produit *Comp*⊗*Canal*⊗*Client*

Définition 3.6 (*Compatibilité 1*)**.** *Deux automates d'interface A_1 et A_2 non vide sont* compatibles (*noté $A_1 \sim A_2$*) *ssi ils ne sont pas vides, composables, et il existe un environnement légal pour A_1 et A_2.*

La composition de A_1 et A_2 est la restriction de leur produit à l'ensemble des états appelés *compatibles* (noté $Comp(A_1,A_2)$). Ce sont les états dans lesquels la composition des deux automates peut se situer sans avoir le risque d'atteindre les états illégaux en activant des actions de sortie ou internes parce qu'elles sont localement contrôlables par l'automate (les actions dans $\Sigma^{\text{loc}}_{A_1 \otimes A_2}$).

Définition 3.7 (*États compatibles*)**.** *Un état (s_1, s_2) est* compatible *dans le produit $A_1 \otimes A_2$ de deux automates d'interface composables A_1 et A_2 s'il n'existe pas d'état $(s'_1, s'_2) \in$ Illégaux(A_1, A_2) atteignable de manière autonome à partir de (s_1, s_2). i.e., il n'existe aucune exécution σ autonome dont l'état initial est (s_1, s_2) et l'état final est (s'_1, s'_2)).*

Définition 3.8 (*Compatibilité 2*)**.** *A_1 et A_2 sont* compatibles *($A_1 \sim A_2$) ssi ils sont composables et l'état initial du produit $A_1 \otimes A_2$ est compatible. Nous notons que la relation de compatibilité $\sim$ est symétrique.*

La composition de deux automates d'interface compatibles est obtenue en limitant le produit à l'ensemble des états compatibles. Tous les états incompatibles et non atteignables sont supprimés du produit.

Définition 3.9 (*Composition $\|$*)**.** *La* composition *$A_1 \parallel A_2$ est définie par*

- $S_{A_1 \| A_2} = Comp(A_1, A_2)$;
- $i_{A_1 \| A_2} = i_{A_1 \otimes A_2}$;
- $\Sigma^{\text{I}}_{A_1 \| A_2} = \Sigma^{\text{I}}_{A_1 \otimes A_2}$, $\Sigma^{\text{O}}_{A_1 \| A_2} = \Sigma^{\text{O}}_{A_1 \otimes A_2}$, *et* $\Sigma^{\text{H}}_{A_1 \| A_2} = \Sigma^{\text{H}}_{A_1 \otimes A_2}$;

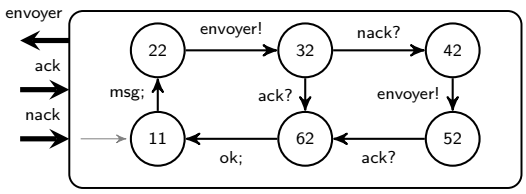


FIGURE 3.5 – La composition *Comp*‖*Client*

– $\delta_{A_1\|A_2} = \delta_{A_1\otimes A_2} \cap (Comp(A_1, A_2) \times \Sigma_{A_1\|A_2} \times Comp(A_1, A_2))$.

3.4 Algorithme de calcul des états compatibles

L'algorithme de vérification de compatibilité [dAH01] entre deux automates d'interface composables décide s'ils sont compatibles en calculant l'ensemble $Comp(A_1,A_2)$ et de vérifier s'il est non vide. En effectuant un parcours en arrière à partir des états illégaux de $A_1 \otimes A_2$, l'algorithme marque les états sources des transitions étiquetées par des actions de sortie et internes jusqu'à la première action d'entrée rencontrée. Avant de présenter l'algorithme, on introduit l'opérateur $OHpre_A(S')$ où A est un automate d'interface et $S' \subseteq S_A$. $OHpre_A(S')$ contient les états de A prédécesseurs des états appartenant à S' en activant uniquement les transition étiquetées par des actions localement contrôlables dans $\Sigma^{\text{loc}}_{A_1\otimes A_2}$. Formellement l'opérateur $OHpre_A : 2^{S_A}\ ^1\!\mapsto 2^{S_A}$ est défini pour tout $S' \subseteq S_A$ par

$$OHpre_A(S') = \{r \in S_A \mid (\exists (r, a, s) \in \delta_A \mid a \in \Sigma^{\text{O}}_A \cup \Sigma^{\text{H}}_A \wedge s \in S')\}.$$

Algorithme 1 : CALCULÉTATSCOMPATIBLES

Entrées : Deux automates d'interface A_1 et A_2
Sorties : $Comp(A_1,A_2)$
1 **Initialisation** : Soit $U_0 = Illégaux(A_1,A_2)$
2 **début**
3 | **répéter**
4 | | Pour $k \geq 0$, $U_{k+1} = U_k \cup OHpre_{A_1\otimes A_2}(U_k)$;
5 | **jusqu'à** $U_{k+1} = U_k$;
6 | **retourner** $S_{A_1\otimes A_2} \setminus U_k$
7 **fin**

L'ensemble $Comp(A_1,A_2)$ est calculable en itérant l'opérateur $OHpre_{A_1\otimes A_2}$ à partir des états illégaux dans $Illégaux(A_1,A_2)$. Les états compatibles sont calculables en temps linéaire $\mathbf{O}(n_1 \times n_2)$ avec n_1 et n_2 les nombres des transitions atteignables dans le produit.

Nous pouvons donc définir l'ensemble des transitions $\delta_{A_1\|A_2}$ de $A_1 \parallel A_2$ en se basant sur cet algorithme. $A_1 \parallel A_2$ est obtenu à partir de $A_1 \otimes A_2$ en enlevant les transitions $(s, a, s') \in \delta_{A_1\otimes A_2}$ telles que

1. $s \in Comp(A_1,A_2)$;

1. 2^S est l'ensemble des parties d'un ensemble S.

2. $a \in \Sigma^{\mathrm{I}}_{A_1 \otimes A_2}$;
3. $s' \notin \ Comp(A_1,A_2)$;

Exemple 3.4. Les deux automates d'interface *Client* et *Comp* sont compatibles car l'état initial de leur produit est compatible. La composition *Comp*‖*Client* est la restriction du produit *Comp*⊗*Client* aux états compatibles (les états atteignables dans le produit *Comp*⊗*Client* privé l'état (7,2)). La figure 3.5 représentent l'automate d'interface *Comp*‖*Client*. ■

Théorème 3.1. *Soient trois automates d'interface A_1, A_2 et A_3 mutuellement composables, alors les automates d'interface $(A_1 \parallel A_2) \parallel A_3$ et $A_1 \parallel (A_2 \parallel A_3)$ sont égaux.*

La preuve de ce théorème est détaillée dans l'annexe A. Le théorème d'associativité peut être généralisé pour $n > 0$ automates d'interface composables $A_1, ..., A_n$. Les automates $A_1, ..., A_n$ sont compatibles si leur composition $A_1 \parallel ... \parallel A_n$, construite de manière graduelle et incrémentale (construire $(A_1 \parallel ... \parallel A_{i-1}) \parallel A_i$ pour tout $i \ \in 1, 2, ..., n$) et dans n'importe quel ordre, n'est pas vide.

3.5 Raffinement

L'objectif de la relation du raffinement pour vérifier la substitution d'un composant par une version plus évoluée. Dans l'approche *input-enabled*, le raffinement est défini par une sorte de simulation ou contenance de traces. Cette configuration est basée sur l'hypothèse que le comportement de la version raffinée d'un composant est inclus dans celui de l'abstraction. Dans l'approche *non-input-enabled* des automates d'interface, cette propriété n'est pas satisfaite. Si les services offerts ou les entrées d'un composant raffiné sont inclus dans les entrées de son abstraction, alors le raffinement peut être utilisé dans un environnement plus restrictif que l'abstraction. Dans le contexte des composants, le raffinement d'un composant doit offrir plus de services (plus d'entrées) et demande moins (moins de sorties). Selon cette contrainte, le raffinement peut être utilisé dans un environnement plus grand que son abstraction et il doit être défini de manière à supporter plus d'entrée et moins de sorties.

Formellement, le raffinement des automates d'interface doit être défini en tant qu'une *simulation alternée* [AHKV98]. Un automate d'interface A' raffine un autre A si toutes les transitions étiquetées par des actions d'entrée de A sont simulées par celles de A' et toutes les transitions étiquetées par des actions de sortie de A' sont simulées par celles de A. Les actions internes ne doivent pas être pris en compte par la relation de raffinement. Pour définir formellement le raffinement, nous introduisons un ensemble de préliminaires introductifs.

Considérons un automate d'interface A et deux états s_1 et $s_2 \ \in \ S_A$, nous définissons les relations suivantes :

- $s_1 \xrightarrow{a}_A s_2$ ssi $(s_1,a,s_2) \ \in \delta_A$;
- $s_1 \xrightarrow{\tau}_A s_2$ ssi $s_1 \xrightarrow{b}_A s_2$ tel que $b \ \in \Sigma^{\mathrm{H}}_A$;
- $s_1 \overset{\varepsilon}{\Longrightarrow}_A s_2$ ssi $s_1 (\xrightarrow{\tau}_A)^* s_2$ tels que * est la juxtaposition réflexive et transitive des transitions.

La fermeture *ε-fermeture* d'un état s est l'ensemble de tous les états atteignables à partir de s en activant uniquement des transitions étiquetées par des actions internes. L'environnement ne peut distinguer entre s et tous les états appartenant dans l'ensemble *ε-fermeture*(s). *ε-fermeture*$_A(s)$ est le plus petit ensemble $R \subseteq S_A$ tel que (1) $s \ \in R$ et (2) pour tout $s' \in R$, s'il existe s'' tel que $s' \overset{\varepsilon}{\Longrightarrow}_A s''$, alors $s'' \in R$.

Un automate d'interface A doit accepter une action de sortie a à partir de l'environnement si a est activable à partir de chaque état appartenant à ε-*fermeture*(s). Contrairement, A peut émettre une action de sortie b au moins à partir d'un seul état dans ε-*fermeture*(s) vers l'environnement. En se basant sur ces intuitions, nous définissons, pour chaque état s, les deux ensembles $ExtActv^{\mathrm{O}}_A(s) = \{a \mid (\exists\ r \in \varepsilon\text{-}fermeture(s) \mid a \in \Sigma^{\mathrm{O}}_A(r))\}$ et $ExtActv^{\mathrm{I}}_A(s) = \{a \mid (\forall\ r \in \varepsilon\text{-}fermeture(s) \mid a \in \Sigma^{\mathrm{I}}_A(r))\}$. Soit un état $s \in S_A$ et une action $a \in \Sigma_A$, l'ensemble $Succ_A(s, a)$ est défini par $\{s' \in S_A \mid (s, a, s') \in \delta_A\}$.

En se basant sur les éléments précédemment introduits, la simulation alternée des automates d'interface est formellement définie comme suit.

Définition 3.10 (*Simulation alternée* $\preceq$)**.** *Soient deux automates d'interface A et A', la relation binaire $\preceq \subseteq S_{A'} \times S_A$ entre A' et A est dite* simulation alternée *ssi pour toute action $s \in S_A$ et $r \in S_{A'}$ tels que $r \preceq s$ les conditions suivantes sont satisfaites :*

1. *pour toute action $a \in \Sigma^{\mathrm{I}}_A(s)$ et $s' \in Succ_A(s, a)$, il existe un état $r' \in Succ_{A'}(r, a)$ tel que $r' \preceq s'$;*
2. *pour toute action $a \in \Sigma^{\mathrm{O}}_{A'}(r)$ et $r' \in Succ_{A'}(r, a)$, il existe un état $s'' \in \varepsilon$-$fermeture(s)$ et un état $s' \in Succ_A(s'', a)$ tel que $r' \preceq s'$;*
3. *pour toute action $a \in \Sigma^{\mathrm{H}}_{A'}(r)$ et $r' \in Succ_{A'}(r, a)$, il existe un état $s' \in \varepsilon$-$fermeture(s)$ tel que $r' \preceq s'$.*

Les deux premières conditions de la définition expriment la dualité entre les actions d'entrée et sortie entre les états r et s : les transitions étiquetées par des actions d'entrée activables à partir de s doivent être activables aussi à partir de r. Chaque transition activable à partir de r et étiquetée par une action de sortie a doit être simulée par une séquence de taille 0 ou $n \geq 1$ transitions internes commençant à partir de s et suivie par une transition étiquetée par l'action a. La troisième condition indique que chaque transition interne activable à partir de r peut être simulée par une séquence de zéro ou plusieurs transition internes commençant à partir de s. Si $r \preceq s$, alors la propriété suivante est satisfaite :

$$ExtActv^{\mathrm{I}}_A(s) \subseteq ExtActv^{\mathrm{I}}_{A'}(r) \wedge ExtActv^{\mathrm{O}}_{A'}(r) \subseteq ExtActv^{\mathrm{O}}_A(s)$$

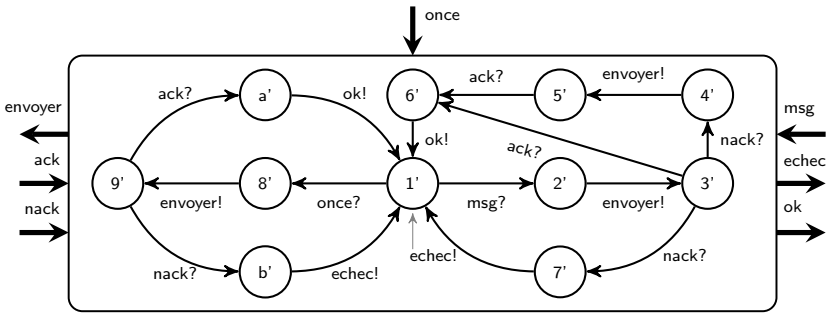


FIGURE 3.6 – Raffinement de l'automate d'interface *Comp*

Définition 3.11 (*Raffinement*)**.** *Le* raffinement *A' d'un automate d'interface A, noté $A' \preceq A$, satisfait les conditions suivantes :*

- $\Sigma^{\mathrm{I}}_{A} \subseteq \Sigma^{\mathrm{I}}_{A'}$ *et* $\Sigma^{\mathrm{O}}_{A} \supseteq \Sigma^{\mathrm{O}}_{A'}$ *;*
- *Il existe une simulation alternée* $\preceq$ *entre* A' *et* A *tel que* $i_{A'} \preceq i_A$*.*

Exemple 3.5. L'automate d'interface présenté dans la figure 3.6 est un raffinement de l'automate d'interface *Comp*. Cet automate représente un composant qui offre deux services : le premier est celui offert par le composant de *Comp* (cf. figure 3.1) qui donne la possibilité d'envoyer un message *msg* une deuxième fois en cas d'échec et le deuxième donne la possibilité uniquement une seule fois pour envoyer le message *once*. Le raffinement respecte toutes les conditions de la définition 3.10 : il existe une simulation alternée entre s et s' pour tout $s \in \{1, 2, 3, 4, 5, 6, 7\} \subseteq S_{\mathit{Comp}}$. ■

Le raffinement entre automates d'interface peut être vérifié en temps polynomial. Plus précisément, si A et A' sont deux automates d'interface dont les états et les transitions sont tous atteignables, alors $A' \preceq A$ peut être vérifié en temps $\mathbf{O}((|\delta_A| + |\delta_{A'}|) \times (|S_A| + |S_{A'}|))$ [AHKV98].

Théorème 3.2. *Le raffinement des automates d'interface est un pré-ordre : pour tout automates d'interface* A_1*,* A_2 *et* A_3*, nous avons* $A_1 \preceq A_1$ *et si* $A_1 \preceq A_2$ *et* $A_2 \preceq A_3$*, alors* $A_1 \preceq A_3$*.*

Démonstration. La preuve est triviale en se basant sur les deux définitions 3.10 et 3.11. □

Le théorème suivant (preuve détaillée dans l'annexe A) montre que les automates d'interfaces supporte les caractéristique de la substitution : un automate d'interface A peut être substitué par une version plus raffinée A' s'il sont connectés à l'environnement au plus par les mêmes entrées. Le cas où l'environnement fournit des entrées pour A' qui ne sont pas fournies pour A peut provoquer des incompatibilités en traitant ces entrées.

Théorème 3.3. *Soient trois automates d'interface* A*,* A' *et* E *tels que* A' *et* E *sont composables et* $\Sigma^{\mathrm{I}}_{A'} \cap \Sigma^{\mathrm{O}}_{E} \subseteq \Sigma^{\mathrm{I}}_{A} \cap \Sigma^{\mathrm{O}}_{E}$*. Si* $A \sim E$ *et* $A' \preceq A$*, alors* $A' \sim E$ *et* $A' \parallel E \preceq A \parallel E$*.*

3.6 Synthèse

Dans ce chapitre, nous avons introduit le formalisme des automates d'interface sur lequel sont basés nos travaux. Nous avons présenté la définition des automates d'interface et leur approche optimiste d'interopérabilité. Pour effectuer l'assemblage des composants, un algorithme de vérification de la compatibilité est donné. Nous avons également présenté la relation de simulation alternée des automates d'interface utilisée pour garantir la substitution des composants.

Deuxième partie

Nos contributions

Chapitre 4

Interopérabilité sémantique et automates d'interface

L'interopérabilité fonctionnelle des composants est vérifiée aux niveaux signature, sémantique et protocole. Les automates d'interface ne sont pas donc suffisants pour assurer une interopérabilité fiable entre les composants. Dans ce chapitre, nous présentons l'approche optimiste des automates d'interface qui permet de traiter la vérification des trois niveaux de l'interopérabilité fonctionnelle. En effet, nous ajoutons aux automates d'interface des informations sur les signatures et la sémantique des opérations et nous exploitons ces informations pour fiabiliser la vérification de l'interopérabilité.

Dans la section 4.1, nous présentons la définition des automates d'interface dans le cadre de notre approche. Dans la section 4.2, nous décrivons comment vérifier la compatibilité entre les composants, dont les contrats sont spécifiés par des automates d'interface, en exploitant la sémantique décrites par les pré et post-conditions des paramètres des actions. Dans la section 4.3, nous expliquons comment la simulation alternée est adaptée notre approche pour vérifier la substitution des composants. Nous appliquons ces résultats sur un exemple issu du modèle de composants EJB de la plateforme J2EE.

4.1 Définitions

La signature d'une action d'entrée (resp. de sortie) a est la signature de l'opération correspondante implémentée par le composant (resp. sollicitée par le composant). Dans le cas où l'opération est associée à un ensemble de paramètres d'entrée et de sortie, l'action correspondante est de la forme $a(i_1,...,i_n) \rightarrow (o)$. L'ensemble $P_a^{\mathrm{i}} = \{i_1, ..., i_n\}$ représente l'ensemble des paramètres d'entrée de a. L'ensemble des paramètres de sortie P_a^{o} est défini par le singleton $\{o\}$ (nous assumons que les opérations ont au plus une seule valeur de retour). L'ensemble de tous les paramètres d'une action a est représentée par P_a. L'absence des paramètres d'entrée ou de sortie est représentée par $()$. Une action n'a pas de signature dans le cas où elle correspond à l'acte d'envoyer ou de recevoir d'une valeur de retour [1]. La sémantique des actions est représentée par des pré et des post-conditions définies sur les paramètres. Ces conditions sont traduites par des formules de la logique de premier ordre [HA50, Hod01] étendues par des théories contextuelles. Soit un ensemble de variables V, nous notons par $Preds(V)$, l'ensemble des prédicats de premier ordre dont les variables libres appartiennent à V.

1. L'envoi de la valeur de retour est représenté, par exemple en Java, par la commande `return`. La réception de la valeur de retour est traduite par l'affectation de la valeur de retour à une variable intermédiaire.

Nous adaptons la définition des automates d'interface donnée dans la section 3.2 comme suit.

Définition 4.1 (*Automate d'interface 2*)**.** *Un* automate d'interface A *est un tuple* $\langle S_A, i_A, \Sigma_A^{\mathrm{I}}, \Sigma_A^{\mathrm{O}}, \Sigma_A^{\mathrm{H}}, \delta_A, \Psi_A \rangle$ *tels que :*

- S_A *est ensemble fini d'états ;*
- $i_A \in S_A$ *est l'état initial ;*
- $\Sigma_A^{\mathrm{I}}, \Sigma_A^{\mathrm{O}}$ *et* Σ_A^{H} *sont respectivement les ensembles des noms des actions d'entrée, de sortie et internes ;*
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *est l'ensemble des transitions entre les états ;*
- Ψ_A *est une fonction qui associe pour chaque action* $a \in \Sigma_A$ *un tuple* $\langle Pre_{\Psi_A(a)}, Post_{\Psi_A(a)} \rangle$ *tels que* $Pre_{\Psi_A(a)} \in Preds(P_a^{\mathrm{i}})$ *et* $Post_{\Psi_A(a)} \in Preds(P_a^{\mathrm{i}} \cup P_a^{\mathrm{o}})$.

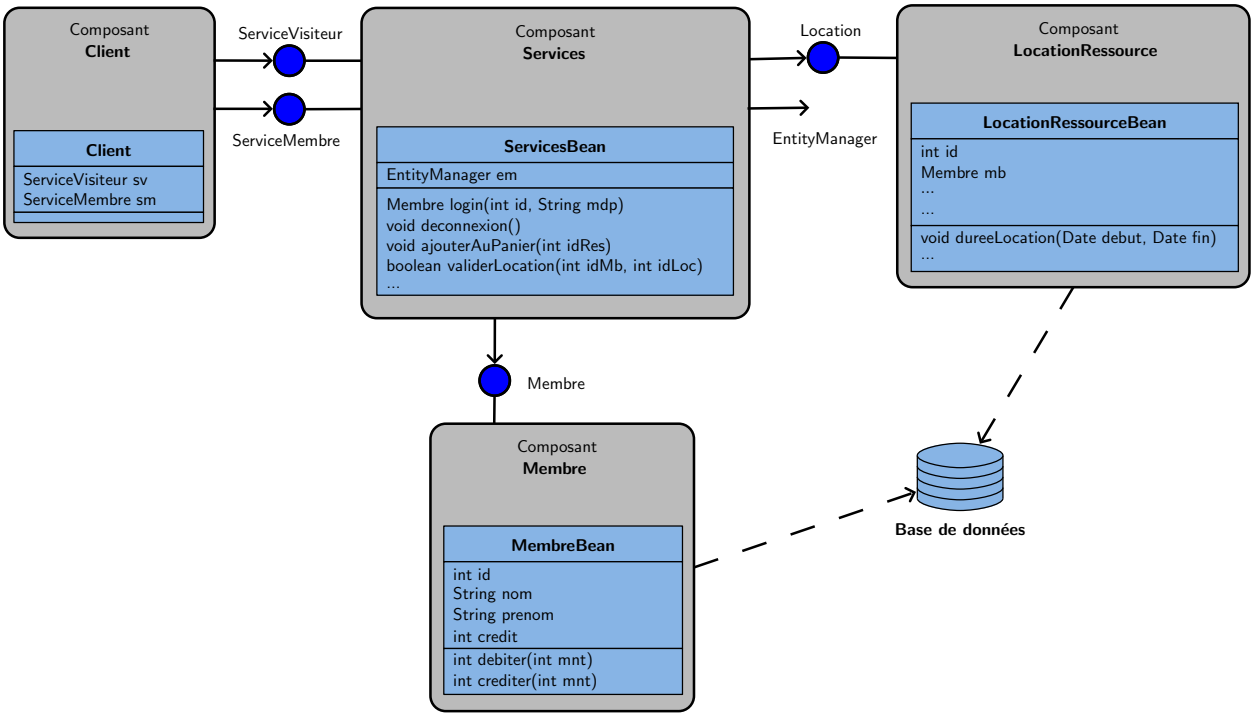


FIGURE 4.1 – Modèle EJB de l'application

Exemple 4.1. Nous présentons une simple application distribuée multi-tiers implémentée par le modèle de composants Entreprise JavaBeans. L'application permet de fournir des services de locations d'objets entre utilisateurs. Le composant Services est un composant *session bean* coté serveur d'application qui joue le rôle d'un médiateur entre le Client et les composants *entity beans*. Les composants *entity beans* représentent la vue orientée objet des lignes de la base de données (cf. section 1.7.1). Une instance du composant Membre représente un client inscrit dans l'application. Une instance du composant LocationRessource représente une location effectuée par le client. Nous n'allons pas donner des détails sur l'implémentation des EJBs en Java, mais nous allons expliquer tout simplement les détails nécessaires à notre exemple.

L'interface `ServiceVisiteur` du composant Services contient la méthode `login` qui retourne une référence à une instance du composant *entity bean* Membre. Cette instance représente l'utilisateur membre après la connexion. La requête envoyée par le visiteur pour se connecter est

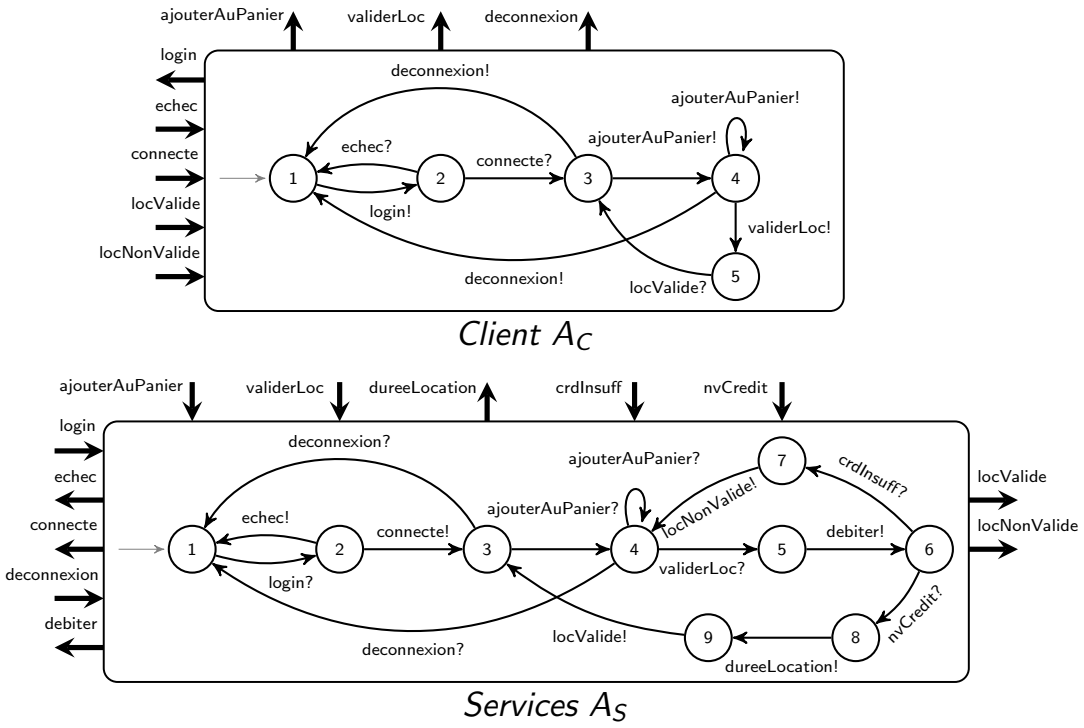


FIGURE 4.2 – Automates d'interface A_C et A_S des composants Client et Services

représentée par l'action de sortie *login* !. Le composant Services provoque une exception *echec* ! en cas d'échec de connexion. L'interface `ServiceMembre` du composant Services contient les méthodes `ajouterAuPanier`, `validerLocation` et `deconnexion`. En invoquant la première méthode (*ajouterAuPanier* !), l'utilisateur ajoute un nouvel objet, représenté par une instance d'un composant *entity bean* Ressource, dans le panier virtuel. La deuxième (*validerLoc* !) permet de valider la location des objets mis dans le panier et elle ne peut pas être appelée sans que la méthode *ajouterAuPanier* soit appelée au minimum une fois. L'opération commence par appeler la méthode `débiter` de l'instance du composant Membre qui permet de débiter un montant du crédit disponible du client. Si le crédit est suffisant, le nouveau montant du crédit est retourné, une durée de location est affectée aux objets loués (*dureeLocation* !) et la location est complètement validée (*locValide* !). Si le crédit est insuffisant, une exception est renvoyée (*crdInsuff* ?) au composant Services. Le composant Services, dans ce cas, provoque une exception *locNonValide* !. Nous assumons que le client s'attend à ce que la validation d'une location doit être achevée avec succès (le client ne traite pas l'exception *locNonValide*). L'action *deconnecter* ! est appelée par le client pour finir la session en se déconnectant. La figure 4.1 présente une représentation abstraite du modèle. La figure 4.2 représente les automates d'interface des composants Client et Services. Nous indiquons que les automates présentés montre uniquement les transitions et les actions d'entrée et de sortie. Aucune information sur la sémantique n'est donnée pour ne pas alourdir les dessins par les pré et post-conditions.

L'ordre des actions décrit par les automates d'interface est implicite. Il est traduit par la mise à jour des attributs privés de chaque composant. Par exemple, un attribut `session_active` doit être positionné à vrai après l'exécution de l'opération `login`. Le même attribut est remis à faux après la déconnexion. Les opérations `ajouterAuPanier` et `validerLocation` ne peuvent pas être

exécutées si l'attribut `session_active` est positionné à faux.

Client A_C	*Services* A_S
$Pre_{\Psi_{A_C}(login)} \equiv id > 0 \wedge 8 \leq mdp.length() \leq 10$	$Pre_{\Psi_{A_S}(login)} \equiv id \geq 1 \wedge 6 \leq mdp.length() \leq 10$
$Post_{\Psi_{A_C}(login)} \equiv membre.getId() = id$	$Post_{\Psi_{A_S}(login)} \equiv membre.getId() = id$
$Pre_{\Psi_{A_C}(echec)} \equiv errmess.length() \neq 0$	$Pre_{\Psi_{A_S}(echec)} \equiv errmess.length() \neq 0$
$Post_{\Psi_{A_C}(echec)} \equiv$ *vrai*	$Post_{\Psi_{A_S}(echec)} \equiv$ *vrai*
$Pre_{\Psi_{A_C}(ajouterAuPanier)} \equiv idRes \neq 0$	$Pre_{\Psi_{A_S}(ajouterAuPanier)} \equiv idRes \neq 0$
$Post_{\Psi_{A_C}(ajouterAuPanier)} \equiv$ *vrai*	$Post_{\Psi_{A_S}(ajouterAuPanier)} \equiv$ *vrai*
$Pre_{\Psi_{A_C}(validerLoc)} \equiv idMb \neq 0 \wedge idLoc \neq 0$	$Pre_{\Psi_{A_S}(validerLoc)} \equiv idMb \neq 0 \wedge idLoc \neq 0$
$Post_{\Psi_{A_C}(validerLoc)} \equiv valide =$ *vrai*	$Post_{\Psi_{A_S}(validerLoc)} \equiv valide =$ *vrai*
$Pre_{\Psi_{A_C}(locNonValide)} \equiv errmess.length() \neq 0$	$Pre_{\Psi_{A_S}(locNonValide)} \equiv errmess.length() \neq 0$
$Post_{\Psi_{A_C}(locNonValide)} \equiv$ *vrai*	$Post_{\Psi_{A_S}(locNonValide)} \equiv$ *vrai*

TABLE 4.1 – Sémantique des actions *login*, *echec*, *ajouterAuPanier*, *validerLoc*, et *locNonValide*

Les actions partagées entre Client et Services, dont les signatures sont définies, sont les suivantes : *login*(*id*,*mdp*) →(*membre*), *deconnexion*()→(), *ajouterAuPanier*(*idRes*)→(), *validerLoc* (*idMb*,*idLoc*)→(*valide*), *echec*(*errmess*)→() et *locNonValide*(*errmess*)→(). Les signatures des actions *connecte* et *locValide* ne sont pas définies car elles correspondent à des valeurs de retour. Les paramètres *mdp* de l'action *login* et *errmess* des actions *echec* et *locNonValide* sont de type chaîne de caractères. Les paramètres d'entrée *id*, *idRes*, *idMb* et *idLoc* des actions *login*, *ajouterAuPanier* et *validerLoc* sont des entiers. Le paramètre de sortie *valide* de l'action *validerLoc* est un booléen. Le type du paramètre de sortie *membre* de l'action *login*, qui est une référence d'objet du composant *entity bean* Membre, est une structure de données.

Le tableau 4.1 présente les sémantiques des actions partagées *login*, *ajouterAuPanier*, *validerLoc*, *echec* et *locNonValide*. Les pré et post-conditions des actions partagées restantes sont positionnées à *vrai*. ■

4.2 Composabilité, compatibilité et composition

Nous rappelons que $Partagées(A_1,A_2) = (\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}) \cup (\Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A_1})$ est l'ensemble des actions partagées par A_1 et A_2.

Définition 4.2 (*Composabilité 2*)**.** *Deux automates d'interface A_1 et A_2 sont* composables *ssi*

- $\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{I}}_{A_2} = \Sigma^{\mathrm{O}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2} = \Sigma^{\mathrm{H}}_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^{\mathrm{H}}_{A_2} = \emptyset$;
- *pour tout* $a \in Partagées(A_1,A_2)$ *dont la signature est donnée par* $a(i_1,...,i_n) \rightarrow (o)$ *dans* A_1 *et par* $a(i'_1,...,i'_n) \rightarrow (o')$ *dans* A_2 *pour* $n \in \mathbb{N}^*$
  - *si* $a \in \Sigma^{\mathrm{O}}_{A_1}$, *alors* $D_{i_k} \subseteq D_{i'_k}$ *pour* $1 \leq k \leq n$ *et* $D_o \subseteq D_{o'}$;
  - *si* $a \in \Sigma^{\mathrm{I}}_{A_1}$, *alors* $D_{i_k} \supseteq D_{i'_k}$ *pour* $1 \leq k \leq n$ *et* $D_o \supseteq D_{o'}$.

La deuxième condition de la définition représente la propriété de sous-typage *des paramètres des actions partagées expliquée dans la section 2.3.1.*

La composition de deux automates d'interface ne peut prendre effet que si les conditions suivantes sont satisfaites : (i) les ensembles de leurs actions d'entrée sont disjoints et de même pour les actions de sortie, (ii) l'ensemble des actions internes de l'un des deux automates est disjoint

avec l'ensemble des actions de l'autre, et (ii) la propriété de sous-typage (cf. section 2.3.1) des domaines des paramètres des actions partagées, appartenant à $Partagées(A_1,A_2)$, est satisfaite.

Exemple 4.2. L'ensemble $Partagées(A_C,A_S)$ égal à {*login*, *echec*, *connecte*, *ajouterAuPanier*, *validerLoc*, *locValide*, *locNonValide*, *deconnexion*}. Les deux automates d'interface A_C et A_S sont composables car les deux conditions de la définition 4.2 sont satisfaites. Les propriétés de sous-typage des paramètres est satisfaites par toutes les actions appartenant à l'ensemble $Partagées(A_C,A_S)$. ■

Avant de définir le produit synchrone de deux automates d'interface dans le contexte de cette approche, nous devons réaliser les correspondances entre les noms des paramètres des actions partagées en effectuant l'opération de *renommage*. Le renommage des paramètres consiste à donner les mêmes noms des paramètres d'une action partagée dans ses pré et post-conditions dans A_1 et A_2. Ce traitement est nécessaire pour définir la compatibilité sémantique des actions partagées.

Définition 4.3 (*Renommage des paramètres*)**.** *Soit une action a dans $Partagées(A_1, A_2)$, la signature d'une action a est définie par $a(i_1,...,i_n)\rightarrow(o)$ dans A_1 et par $a(i'_1,...,i'_n)\rightarrow(o')$ dans A_2. Le* renommage *des paramètres dans les sémantiques $\Psi_a^{A_1}$ et $\Psi_a^{A_2}$ est la substitution de i'_1 par i_1,..., i'_n par i_n et o' par o dans $Pre_{\Psi_{A_2}(a)}$ et $Post_{\Psi_{A_2}(a)}$ ou l'inverse dans $Pre_{\Psi_{A_1}(a)}$ et $Post_{\Psi_{A_1}(a)}$.*

Exemple 4.3. Dans notre exemple, on suppose que les noms des paramètres dans les sémantiques des actions dans $Partagées(A_C,A_S)$, sont les mêmes. ■

Nous notons par $\Psi_{A_1/A_2}(a)$, la sémantique de a après le renommage des paramètres respectivement dans A_1 et A_2. Nous pouvons maintenant définir proprement les notions de la compatibilité sémantique des actions partagées.

Définition 4.4 (Compatibilité sémantique)**.** *Soit une action $a \in Partagées(A_1,A_2)$, si l'une des conditions suivantes est vraie alors l'action a dans A_1 est* sémantiquement compatible *avec l'action a dans A_2 – notée par $SemComp_a(A_1, A_2) \equiv$ vrai (faux dans le cas contraire) :*

- *si $a \in \Sigma^{\mathrm{O}}_{A_1}$, alors $Pre_{\Psi_{A_1/A_2}(a)} \Rightarrow Pre_{\Psi_{A_2/A_1}(a)} \wedge Post_{\Psi_{A_2/A_1}(a)} \Rightarrow Post_{\Psi_{A_1/A_2}(a)}$,*
- *si $a \in \Sigma^{\mathrm{I}}_{A_1}$, alors $Pre_{\Psi_{A_2/A_1}(a)} \Rightarrow Pre_{\Psi_{A_1/A_2}(a)} \wedge Post_{\Psi_{A_1/A_2}(a)} \Rightarrow Post_{\Psi_{A_2/A_1}(a)}$.*

Exemple 4.4. Toutes les actions partagées dans $Partagées(A_C, A_S)$ sont sémantiquement compatibles entres les automates d'interface A_C et A_S. ■

Durant la composition de deux automates d'interface, les actions partagées sémantiquement compatibles se synchronisent et les actions non partagées sont entrelacées de façon asynchrone. Nous adaptons la définition du produit synchronisé des automates d'interface de la manière suivante.

Définition 4.5 (*Produit synchronisé 2 $\otimes$*)**.** *Soient A_1 et A_2 deux automates d'interface composables, le* produit synchronisé *$A_1 \otimes A_2$ de A_1 et A_2 est défini par :*

- $S_{A_1\otimes A_2} = S_{A_1} \times S_{A_2}$ *et* $i_{A_1\otimes A_2} = (i_{A_1}, i_{A_2})$ *;*
- $\Sigma^{\mathrm{I}}_{A_1\otimes A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus Partagées(A_1, A_2)$ *;*
- $\Sigma^{\mathrm{O}}_{A_1\otimes A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus Partagées(A_1, A_2)$ *;*
- $\Sigma^{\mathrm{H}}_{A_1\otimes A_2} = \Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2} \cup \{a \in Partagées(A_1, A_2) \mid SemComp_a(A_1, A_2) \equiv \mathsf{vrai}\}$ *;*
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1\otimes A_2}$ *si*

- $a \notin$ *Partagées*$(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$ *ou*
- $a \notin$ *Partagées*$(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$ *ou*
- $a \in$ *Partagées*$(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge$ *SemComp*$_a(A_1, A_2) \equiv$ *vrai* ;

– $\Psi_{A_1 \otimes A_1}$ *est définie par :*
  - Ψ_{A_1} *pour* $a \in \Sigma_{A_1} \setminus$ *Partagées*(A_1, A_2) ;
  - Ψ_{A_2} *pour* $a \in \Sigma_{A_2} \setminus$ *Partagées*(A_1, A_2) ;
  - $\langle Pre_{\Psi_{A_1}(a)}, Post_{\Psi_{A_2}(a)} \rangle$ *pour* $a \in$ *Partagées*$(A_1, A_2) \cap \Sigma^{\mathrm{O}}_{A_1}$ *telle que* *SemComp*$_a(A_1, A_2) \equiv$ *vrai* ;
  - $\langle Pre_{\Psi_{A_2}(a)}, Post_{\Psi_{A_1}(a)} \rangle$ *pour* $a \in$ *Partagées*$(A_1, A_2) \cap \Sigma^{\mathrm{I}}_{A_1}$ *telle que* *SemComp*$_a(A_1, A_2) \equiv$ *vrai* ;

Toute action partagée a où *SemComp*$_a(A_1, A_2) \equiv$ *faux* ne sera jamais activable dans le produit $A_1 \otimes A_2$ de deux automates d'interface composables A_1 et A_2. Ceci traduit son absence dans l'ensemble des actions internes $\Sigma^{\mathrm{H}}_{A_1 \otimes A_2}$.

Théorème 4.1. *Le produit $\otimes$ des automates d'interface composables mutuellement est une opération commutative et associative.*

Dans le cadre de notre approche, le théorème précédent reste vrai. La preuve est détaillée dans l'annexe A.

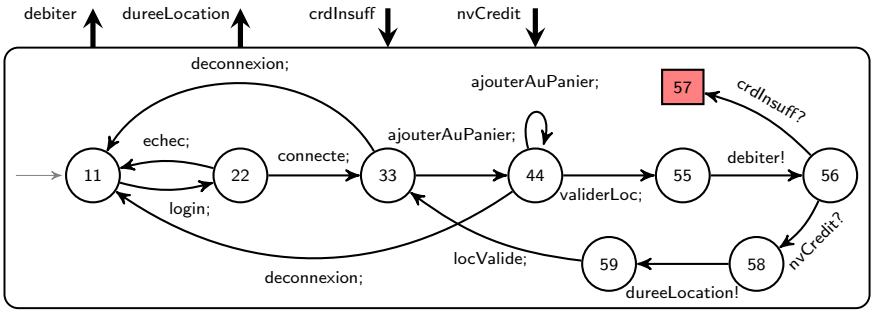


FIGURE 4.3 – $A_C \otimes A_S$ sans les états et les transitions inatteignables

L'incompatibilité entre deux automates d'interface A_1 et A_2 pourrait se produire à cause de (i) l'existence des états (s_1, s_2) dans le produit $A_1 \otimes A_2$ tel qu'il existe au moins une action a dans *Partagées*(A_1,A_2) activable à partir de s_1 et elle ne l'est pas à partir de s_2 ou vice-versa, ou (ii) l'action a est activable à partir de s_1 et s_2 mais *SemComp*$_a(A_1, A_2) \equiv$ *faux*. Ces états sont alors illégaux dans le produit $A_1 \otimes A_2$.

Définition 4.6 (*États illégaux 2*)**.** *L'ensemble des* états illégaux *Illégaux*$(A_1,A_2) \subseteq S_{A_1} \times S_{A_2}$ *dans* $A_1 \otimes A_2$ *est défini par* $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid (\exists\, a \in$ *Partagées*$(A_1, A_2) \mid$ *la condition* $C_1 \vee C_2$ *est satisfaite*$)\}$

$$C_1 = \left(\begin{array}{c} (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \notin \Sigma^{\mathrm{I}}_{A_2}(s_2)) \vee (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \in \Sigma^{\mathrm{I}}_{A_2}(s_2) \\ \wedge\ SemComp_a(A_1, A_2) \equiv \textit{faux}) \end{array} \right)$$

$$C_2 = \left(\begin{array}{c} (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \notin \Sigma^{\mathrm{I}}_{A_1}(s_1)) \vee (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \in \Sigma^{\mathrm{I}}_{A_1}(s_1) \\ \wedge\ SemComp_a(A_1, A_2) \equiv \textit{faux}) \end{array} \right)$$

Exemple 4.5. Sachant que la condition $SemComp_a(A_C, A_S)$ est vraie pour toute action $a \in Partagées(A_C,A_S)$, le produit $A_C \otimes A_S$ des deux automates d'interface A_C et A_S, sans les états et les transitions inatteignables, est donné dans la figure 4.3. L'état (5,7) est un état illégal dans le produit parce que l'action partagée *locNonValide* est activable en sortie à l'état 7 dans A_S et non activable en entrée à l'état 5 dans A_C. ■

Les définitions des états compatibles et la compatibilité de deux automates d'interface, dans le cadre de notre proposition, sont exactement les mêmes que celles données dans le chapitre 3 (cf. les définitions 3.5, 3.6, 3.7 et 3.8).

Définition 4.7 (*Composition 2* $\|$)**.** *La* composition $A_1 \parallel A_2$ *est définie par :*

- $S_{A_1\|A_2} = Comp(A_1, A_2)$;
- $i_{A_1\|A_2} = i_{A_1\otimes A_2}$;
- $\Sigma^{\mathrm{I}}_{A_1\|A_2} = \Sigma^{\mathrm{I}}_{A_1\otimes A_2}$, $\Sigma^{\mathrm{O}}_{A_1\|A_2} = \Sigma^{\mathrm{O}}_{A_1\otimes A_2}$, *et* $\Sigma^{\mathrm{H}}_{A_1\|A_2} = \Sigma^{\mathrm{H}}_{A_1\otimes A_2}$;
- $\Psi_{A_1\|A_2} = \Psi_{A_1\otimes A_2}$;
- $\delta_{A_1\|A_2} = \delta_{A_1\otimes A_2} \cap (Comp(A_1, A_2) \times \Sigma_{A_1\|A_2} \times Comp(A_1, A_2))$.

Exemple 4.6. En appliquant l'algorithme de calcul des états compatibles présenté la section 3.4, nous pouvons constater que A_C et A_S sont compatibles et leur composition $A_C \parallel A_S$ est définie par l'automate d'interface de la figure 4.4. Les composants Client et Services ne fonctionnent correctement que dans le cas où le crédit du client est supérieur au prix de la location des objets sélectionnés dans le panier. L'estimation que le composant *entity bean* Membre termine l'exécution de la méthode `debiter` avec succès, permet de considérer que les composants Client et Services comme compatibles en empêchant l'accessibilité à l'état illégal (5,7).

Si la condition $SemComp_{validerLoc}(A_C, A_S) \equiv$ *faux* par exemple, alors l'état (4,4) devient illégal et il n'y aura pas d'états compatibles dans le produit $A_C \otimes A_S$. Le lecteur peut donc constater que l'insatisfaction de la compatibilité sémantique entre les actions partagées peut provoquer l'incompatibilité entre deux automates d'interface composables. ■

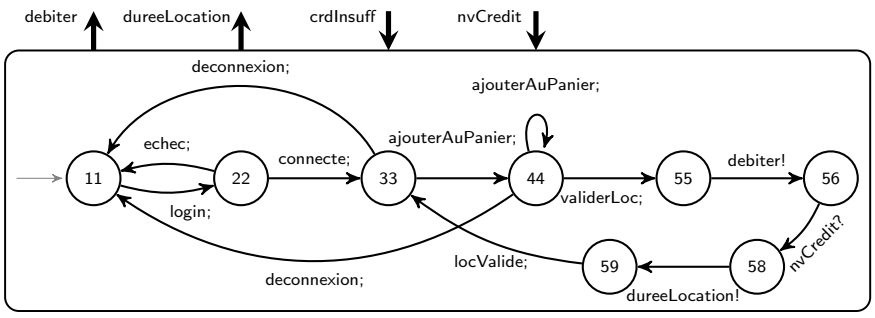


FIGURE 4.4 – Automate d'interface composite $A_C \parallel A_S$

Théorème 4.2. *La composition* $\|$ *des automates d'interface est une opération associative dans le cas où ils sont composables mutuellement.*

Démonstration. Le produit $\otimes$ des automates d'interface composables est associatif en considérant la sémantique des actions. Par conséquent, la preuve est exactement la même que celle du théorème 3.1 sauf que nous devons tenir en compte la définition des états illégaux étendue par la compatibilité sémantique entre les actions partagées (définition 4.6). □

Les étapes de vérification dans cette démarche sont les mêmes que celles décrites dans le chapitre 3, sauf que nous considérons la sémantique des actions durant le calcul du produit et les états illégaux. Durant le calcul du produit, il faut prendre en compte la complexité satisfiabilité des implication entre les pré et post-conditions des actions. L'algorithme 1 proposé dans le chapitre 3 devient donc un semi-algorithme vu la complexité variable des problèmes de satisfiabilité (*satisfiability* (*SAT*) *problems*) de la logique propositionnelle et la logique de premier ordre. Les problèmes de satisfiabilité propositionnelle sont NP-complets [Coo71, Sch78]. Contrairement à la logique propositionnelle, les problèmes de satisfiabilité de la logique de premier ordre sont indécidables.

4.3 Raffinement

Dans cette section, pour garantir les propriétés liées à la substitution des composants, nous adaptons la relation de simulation alternée utilisée pour raffiner les automates d'interface en tenant en compte la sémantique des opérations. Définir la sémantique des actions d'un raffinement d'un automate d'interface doit obéir à un ensemble de contraintes. Premièrement, le raffinement d'un composant peut fournir plus de services aux autres composants et demander moins que sa version d'origine. Deuxièmement, la sémantique des actions dans le raffinement et l'abstraction doit être définie de telle sorte que les propriétés de sous-typage comportemental présentées dans la section 2.3.2 soient satisfaites. Dans la version raffinée d'un automate d'interface, une action d'entrée a doit avoir une précondition plus faible et une post-condition plus forte que celles de l'action dans l'abstraction. La propriété est inversée pour les actions de sortie. Une action de sortie a doit avoir une précondition plus forte et une post-condition plus faible que celles de l'action dans l'abstraction. La définition suivante illustre la relation de sous-typage sous le nom de la *substitution sémantique* des actions entre deux automates d'interfaces. Cette définition est illustrée par la figure 4.5.

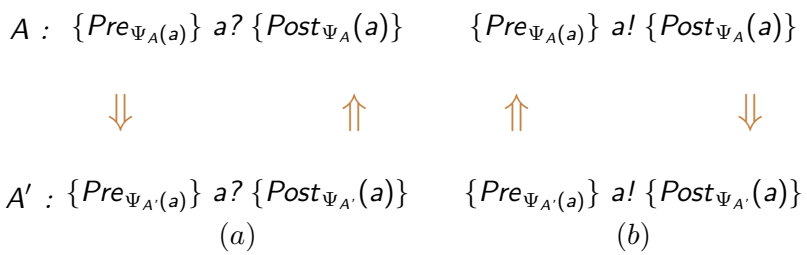


FIGURE 4.5 – Substitution sémantique des actions

Définition 4.8 (*Substitution sémantique*)**.** *Soient deux automates d'interface A et A', une action a dans $\Sigma^{\text{ext}}_{A'}$ peut substituer la même action dans Σ^{ext}_{A} – notée $SemSub_a(A, A') \equiv$ vrai (faux dans le cas contraire), ssi :*

(*a*) $Pre_{\Psi_A(a)} \Rightarrow Pre_{\Psi_{A'}(a)}$ *et* $Post_{\Psi_{A'}(a)} \Rightarrow Post_{\Psi_A(a)}$ *si* $a \in \Sigma^{\text{I}}_A \cup \Sigma^{\text{I}}_{A'}$ *ou*

(*b*) $Pre_{\Psi_A(a)} \Leftarrow Pre_{\Psi_{A'}(a)}$ *et* $Post_{\Psi_{A'}(a)} \Leftarrow Post_{\Psi_A(a)}$ *si* $a \in \Sigma_A^{\mathrm{O}} \cup \Sigma_{A'}^{\mathrm{O}}$.

En se basant sur les préliminaires et la définition de la simulation alternée présentées dans la section 3.5, la définition du raffinement, dans le cadre de notre approche, est adaptée comme suit :

Définition 4.9 (*Raffinement 2*)**.** *Un* raffinement A' *d'un automate d'interface* A*, noté* $A' \preceq A$*, satisfait les conditions suivantes :*

- $\Sigma_A^{\mathrm{I}} \subseteq \Sigma_{A'}^{\mathrm{I}}$ *et* $\Sigma_A^{\mathrm{O}} \supseteq \Sigma_{A'}^{\mathrm{O}}$ *;*
- *Pour toute action* $a \in \Sigma_A^{\mathrm{ext}} \cap \Sigma_{A'}^{\mathrm{ext}}$*,* $SemSub_a(A, A') \equiv$ *vrai ;*
- *Il existe une simulation alternée* $\preceq$ *entre* A' *et* A *tel que* $i_{A'} \preceq i_A$*.*

Selon la définition, toutes les actions externes appartenant à $\Sigma_A^{\mathrm{ext}} \cap \Sigma_{A'}^{\mathrm{ext}}$ doivent respecter la condition de substitution sémantique. Le raffinement entre automates d'interface reste un pré-ordre réflexif et transitif en tenant en compte la propriété de la substitution sémantique.

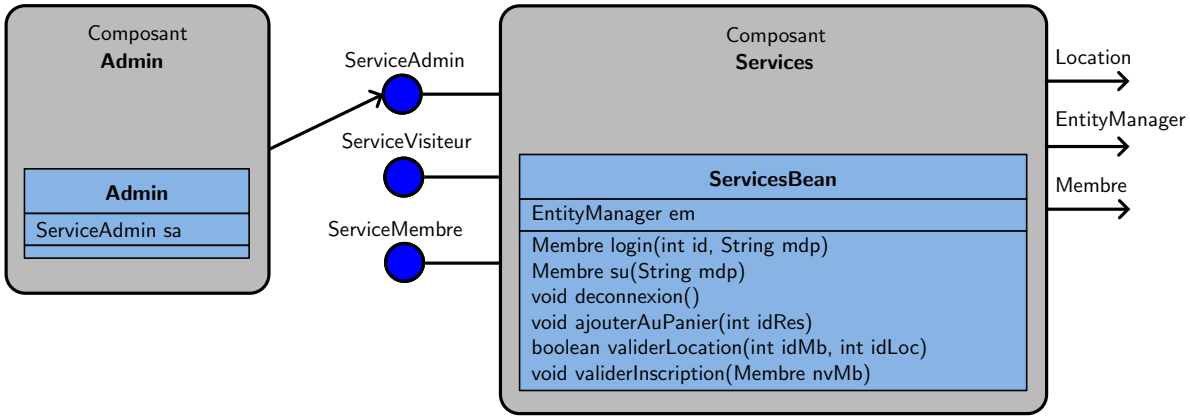


FIGURE 4.6 – Version étendue du composant Services

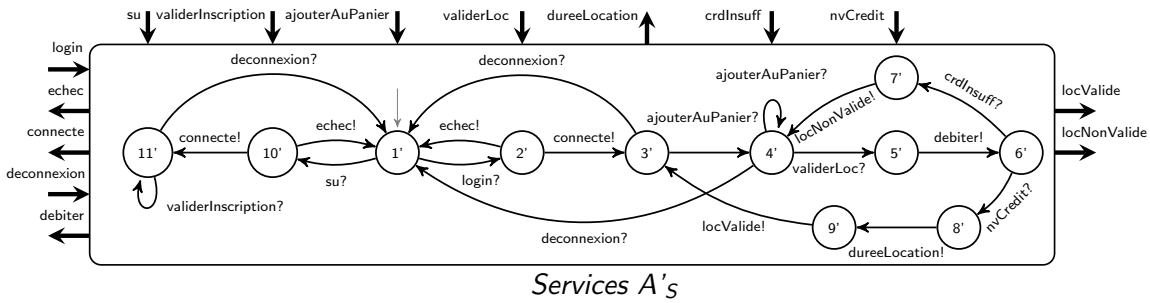


FIGURE 4.7 – Raffinement possible A'_S de A_S

Exemple 4.7. Le composant *session bean* Services peut être raffiné en ajoutant de nouveaux services pour les super-utilisateurs. Ces nouveaux services appartiennent à l'interface `ServiceAdmin`. Le composant implémente les nouvelles méthodes `su` et `validerInscription`. L'administrateur

Admin se connecte par la méthode su qui retourne une référence d'une *entity bean* Membre qui représente l'administrateur après la connexion. En cas d'échec, le composant Services provoque une exception *echec* !. Après l'authentification, l'administrateur peut exécuter pour plusieurs fois l'action *validerInsc* ? pour valider les inscriptions des nouveaux membres ajoutés à la base de données. La figure 4.6 représente une version raffinée du composant Services.

Un raffinement possible de l'automate d'interface A_S, présenté dans la figure 4.2, est donné dans la figure 4.7. Le composant offre deux services, un premier service pour les clients membres et un autre pour les administrateurs. Si pour toute action $a \in \Sigma_A^{ext} \cap \Sigma_{A'}^{ext}$, $SemSub_a(A, A') \equiv vrai$ alors $A'_S \preceq A_S$: (i) A'_S contient plus d'actions d'entrée (*su* et *validerInscription*) et exactement les mêmes actions de sortie de A_S, (ii) il existe une simulation alternée entre s' et s pour tout $s \in S_{A_S}$. ■

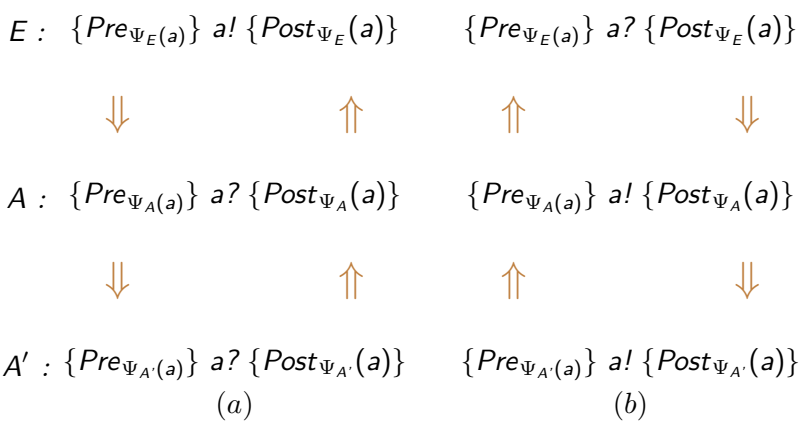


FIGURE 4.8 – Préservation de la compatibilité sémantique des actions par raffinement

La validité de la compatibilité sémantique d'une action externe partagée entre deux automates d'interface composables A et E n'est pas en contradiction avec la validité de la condition de substitution sémantique de a par son équivalente dans A' tel que $A' \preceq A$. Pour toute action $a \in \Sigma_A^I \cap \Sigma_{A'}^I$, si a dans A est sémantiquement compatible avec a dans E, alors a dans A' est sémantiquement compatible avec a dans E (cf. figure 4.8(a)). La propriété est aussi satisfaite pour les actions $a \in \Sigma_A^O \cap \Sigma_{A'}^O$ (cf. figure 4.8(b)).

Propriété 4.1. *Soient trois automates d'interface A, A', et E tels que A et E sont composables, $Partagées(A, E) \neq \emptyset$ et $A' \preceq A$, pour toute action $a \in Partagées(A, E) \cap \Sigma_{A'}^{ext}$, si $SemComp_a(A, E) \equiv vrai$, alors $SemComp_a(A', E) \equiv vrai$.*

Théorème 4.3. *Soient trois automates d'interface A, A' et E tels que A' et E sont composables et $\Sigma_{A'}^I \cap \Sigma_E^O \subseteq \Sigma_A^I \cap \Sigma_E^O$. Si $A \sim E$ et $A' \preceq A$, alors $A' \sim E$ et $A' \| E \preceq A \| E$.*

Démonstration. Nous avons comme hypothèse $A' \preceq A$. Pour toute action $a \in \Sigma_A^{ext} \cap \Sigma_{A'}^{ext}$, $SemSub_a(A, A') \equiv vrai$ (cf. définition 4.9). Cette condition ne contredit pas la preuve du théorème 3.3 dans l'approche d'origine (cf. propriété 4.1). La preuve que $A' \sim E$ est exactement la même que celle du théorème 3.3 sauf que la sémantique des actions est prise en compte lors de la déduction de la compatibilité de A' et E à partir de la compatibilité de A et E.

La preuve que $A' \parallel E \preceq A \parallel E$ est exactement la même que celle du théorème 3.3 sauf qu'il faut prouver en plus que, pour toute action $a \in \Sigma_{A\|E}^{ext} \cap \Sigma_{A'\|E}^{ext}$, $SemSub_a(A \parallel E, A' \parallel E) \equiv vrai$. Cette affirmation est vraie. □

Le corollaire 4.1 représente la propriété de la substitutivité des automates d'interface (*Independent Implementability* [AH05]) qui établit que, sous certaines conditions, les automates d'interface compatibles peuvent être raffinés séparément sans vérifier que leurs raffinements sont compatibles, et la composition de leurs raffinements raffine leur composition.

Corollaire 4.1. *Soient quatre automates d'interface* A_1, A'_1, A_2 *et* A'_2 *tels que*

1. A_1 *et* A_2 *sont composables et* $\Sigma^{\mathrm{I}}_{A'_1} \cap \Sigma^{\mathrm{O}}_{A_2} \subseteq \Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}$ *et*
2. A'_1 *et* A'_2 *sont composables et* $\Sigma^{\mathrm{I}}_{A'_2} \cap \Sigma^{\mathrm{O}}_{A'_1} \subseteq \Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A'_1}$,

si $A_1 \sim A_2$, $A'_1 \preceq A_1$ *et* $A'_2 \preceq A_2$ *alors* $A'_1 \sim A'_2$ *et* $A'_1 \parallel A'_2 \preceq A_1 \parallel A_2$.

Démonstration. Les deux automates d'interface A_1 et A_2 sont composables et $\Sigma^{\mathrm{I}}_{A'_1} \cap \Sigma^{\mathrm{O}}_{A_2} \subseteq \Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}$. Nous avons comme hypothèse $A_1 \sim A_2$ et $A'_1 \preceq A_1$, alors selon le théorème 4.3, (1) $A'_1 \sim A_2$ et (2) $A'_1 \parallel A_2 \preceq A_1 \parallel A_2$.

Également, les automates d'interface A'_1 et A'_2 sont composables et $\Sigma^{\mathrm{I}}_{A'_2} \cap \Sigma^{\mathrm{O}}_{A'_1} \subseteq \Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A'_1}$. Selon le théorème 4.3, le résultat (1) et l'hypothèse $A'_2 \preceq A_2$ impliquent que $A'_1 \sim A'_2$ et (3) $A'_1 \parallel A'_2 \preceq A'_1 \parallel A_2$. Comme la relation $\preceq$ est un pré-ordre, les résultats (2) et (3) impliquent que $A'_1 \parallel A'_2 \preceq A_1 \parallel A_2$. □

4.4 Synthèse

Dans ce chapitre, nous avons présenté une approche formelle basée sur l'approche optimiste des automates d'interface pour vérifier l'interopérabilité des composants. Les actions des automates d'interface sont enrichies par la sémantique des paramètres. Cette sémantique est utilisée pour vérifier la compatibilité des actions partagées de deux automates d'interface. Concernant l'opération de substitution des composants, nous avons adapté la définition du raffinement des automates d'interface décrite par une relation de simulation alternée en exploitant les informations sur la sémantique des actions. L'approche est illustrée par un exemple de composants EJB de la plateforme J2EE.

Chapitre 5

Adaptation sémantique des composants et automates d'interface

L'adaptation logicielle consiste à assurer l'assemblage des composants malgré les disparités qui peuvent exister entre eux (cf. section 2.5). Plusieurs travaux de recherche ont été effectués pour résoudre le problème de l'adaptation au niveau signature. En revanche, les travaux sur l'adaptation aux niveaux sémantique et protocole restent peu développés.

Dans ce chapitre, nous traitons l'adaptation des composants dont les spécifications contractuelles des interfaces sont décrites par des automates d'interface enrichis par la sémantique des opérations. Nous proposons un algorithme de génération automatique d'un automate adaptateur pour deux automates d'interface conformément à un *contrat d'adaptation*. Le contrat d'adaptation est un ensemble de règles reliant certaines opérations non partagées et il représente la spécification la plus abstraite de l'adaptateur. Avant la génération de l'adaptateur, nous vérifions l'adaptabilité sémantique de deux automates d'interface en testant la validité des conditions entre les pré et post-conditions des actions reliées par le contrat d'adaptation. L'originalité de ce travail est la combinaison de l'approche optimiste des automates d'interface, l'intégration du niveau sémantique, et l'adaptation afin d'établir une approche complète d'assemblage et de réutilisation des composants.

Dans la section 5.1, nous détaillons les différences entre notre approche et les approches d'adaptation existantes. Dans la section 5.2, nous décrivons la spécification du contrat d'adaptation entre deux automates d'interface. Dans la section 5.3, nous présentons la notion de l'adaptabilité sémantique entre automates d'interface conformément à un contrat d'adaptation prédéfini. Dans la section 5.4, nous présentons la définition formelle de l'adaptateur de deux automates d'interface conformément au contrat d'adaptation. Dans la section 5.6, nous présentons l'algorithme de construction automatique de l'adaptateur et la preuve de sa correction et de sa complétude.

5.1 Adaptation logicielle et l'approche optimiste

Dans cette section, nous évoquons les différences principales entre notre approche d'adaptation des automates d'interface et les approches existantes. Plusieurs approches existantes sont basées sur des modèles différents des automates d'interface et traitent le problème d'adaptation différemment. À titre d'exemple, l'approche de [CPS06] présentée dans la section 2.5.2 considère que deux ou plusieurs composants soient incompatibles si l'un des composants provoque des sorties "non partagées" qui ne peuvent pas être acceptées par l'un des composants restants. L'approche est donc restrictive car elle suppose que seuls les composants à adapter fournissent

les entrées correspondantes à ces sorties.

En outre, cette approche permet d'adapter plusieurs composants en même temps de manière non incrémentale en assumant que l'environnement de chaque composant soit préfixé auparavant. Elle ne prend pas en compte le dynamisme de communication qui peut avoir lieu au niveau de l'interopérabilité des composants au sein même d'une architecture fixe. Plus précisément, on considère un composant composite qui contient plusieurs sous composants. La communication entre les sous composants ne peut pas avoir une seule configuration et elle peut inclure uniquement une partie d'eux. Par conséquent, plusieurs adaptateurs sont possibles par rapport aux différentes configurations. De plus, cette approche est pessimiste et ne suppose pas qu'il existe un environnement adéquat qui permet d'éviter les situations de blocage.

Une autre approche d'adaptation, comme celle présentée dans [BBC05], est basée sur l'assemblage *binaire* ou *incrémental* des composants. Les adaptateurs sont générés au fur et à mesure durant le processus d'assemblage. Dans l'approche incrémentale, les composants sont assemblés deux par deux et les adaptateurs sont générés automatiquement conformément à des contrats d'adaptation prédéfinis entre les différents composants. Cette approche est plus naturelle et elle s'adapte mieux à la nature des systèmes à base de composants que l'approche non incrémentale. Pour deux composants impliqués dans une communication, un adaptateur est généré conformément à une spécification des relations entre les actions inadéquates. Les entrées et les sorties partagées et celles reliées par l'adaptateur sont synchronisées et les autres sont entrelacées.

En se basant sur les concepts d'adaptation présentés dans la section 2.5, nous rappelons que les problèmes d'inadéquation entre les interfaces des composants surviennent essentiellement dans les cas suivants :

- les noms des opérations appelées ou des messages échangés entre les composants sont différents ; les composants interagissent souvent par les mêmes noms d'opération ;
- l'ordre des actions d'un protocole d'un composant ne respecte pas celui d'un autre ;
- une opération dans un composant n'a pas d'équivalent dans l'autre, ou correspond à plusieurs autres opérations ;
- la sémantique de l'appel d'un service dans un composant n'est pas compatible avec la sémantique des opérations qui correspondent à ce service dans un autre composant.

L'approche d'adaptation présentée dans ce chapitre est basée sur l'approche incrémentale et l'approche optimiste de composition des automates d'interface. Le but est, essentiellement, d'assurer la composition des automates d'interface qui sont censées communiquer via des automates adaptateurs après la spécification des relations entre les actions inadéquates. Contrairement à d'autres approches d'adaptation, les inadéquations entre les actions non partagées ne peuvent pas être détectées en calculant le produit synchronisé de deux automates d'interface composables. Les actions externes qui devraient être synchronisées par l'adaptation sont nommées différemment, ce qui conduit à les rendre absentes dans l'ensemble des actions partagées. En se basant sur la définition 3.3 du produit synchronisé, toutes les actions non partagées ne synchronisent pas et elles sont entrelacées dans le produit.

Dans le cadre de l'approche optimiste, seuls les automates d'interface composables, reliés par des règles d'adaptation définies entre les actions non partagées, pourraient être adaptés. Dans ce cas, le produit entre l'automate adaptateur et les deux automates d'interface est défini différemment et il est soumis à des règles de synchronisation entre les actions qui diffèrent de celles utilisées pour le calcul du produit classique. La procédure de calcul de la composition entre

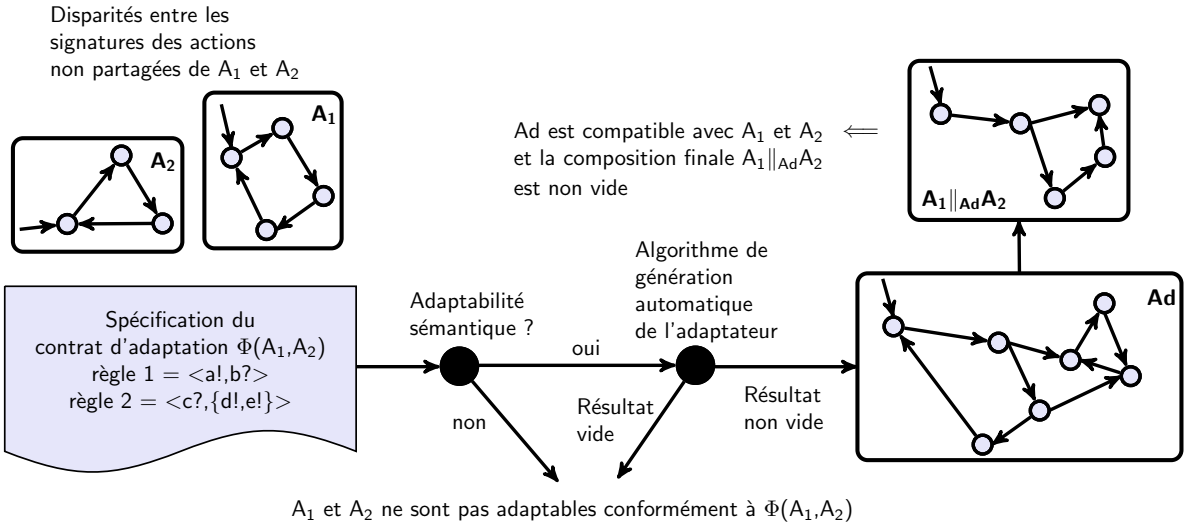


FIGURE 5.1 – Processus d'adaptation de deux automates d'interface composables A_1 et A_2.

les deux automates d'interface après l'adaptation est similaire à celle appliquée sur le produit classique de deux ou plusieurs automates d'interface pour vérifier leur compatibilité. Les règles d'adaptation et leurs propriétés intrinsèques (cf. section 5.2) sont définies par l'assembleur en se basant sur les rôles des actions inadéquates décrits par leurs signatures et leurs sémantiques dans les contrats d'interfaces des deux composants.

Notre approche a pour but de tester si l'adaptation de deux automates d'interface A_1 et A_2 composables assure leur compatibilité conformément aux règles d'adaptation, et, ainsi, générer un adaptateur Ad pour A_1 et A_2 dans le cas où l'adaptation assurera la compatibilité. Plus précisément, notre algorithme proposé dans la section 5.6 permet de détecter s'il y a des états illégaux, dans le produit final $A_1 \circledast_{Ad} A_2$ de l'adaptateur Ad avec A_1 et A_2, qui vont conduire à rendre leur composition finale $A_1 \parallel_{Ad} A_2$ vide. Dans le cas où l'algorithme détecte la présence des chemins autonomes (cf. section 3.2) dans le produit $A_1 \circledast_{Ad} A_2$ entre l'état initial et un état illégal, il génère un adaptateur vide. Dans le cas contraire, il génère un adaptateur qui respecte les règles d'adaptation et assure la compatibilité de A_1 et A_2.

L'adaptation de deux automates d'interface A_1 et A_2, conformément à un contrat d'adaptation $\Phi(A_1, A_2)$ (cf. section 5.2) décrit en fonction des actions non partagées, est effectuée en passant par les étapes suivantes :

1. si A_1 et A_2 sont composables, vérifier l'adaptabilité sémantique entre les actions reliées par le contrat d'adaptation. Si ces actions sont sémantiquement adaptables, alors on passe à la deuxième étape. Sinon, A_1 et A_2 ne peuvent pas être adaptés conformément à $\Phi(A_1, A_2)$ (cf. section 5.3) ;
2. appliquer l'algorithme de génération automatique d'adaptateur (cf. section 5.6) sur A_1 et A_2 et le contrat d'adaptation $\Phi(A_1, A_2)$. L'algorithme génère un adaptateur non vide Ad pour A_1 et A_2 conformément à $\Phi(A_1, A_2)$ si A_1 et A_2 sont adaptables. Dans ce cas, l'automate Ad permet de consommer les sorties de l'un des deux automates avant de générer les sorties pour leurs entrées correspondantes dans l'autre automate. L'automate d'interface du composite final est défini par le produit $A_1 \circledast_{Ad} A_2$. Dans le cas où l'algorithme

retourne un automate vide, les deux automates d'interface A_1 et A_2 ne sont pas adaptables conformément à $\Phi(A_1, A_2)$.

La figure 5.1 illustre le processus d'adaptation dans le cadre de notre approche.

5.2 Contrat d'adaptation

Le *contrat d'adaptation* de deux automates d'interface composables A_1 et A_2 est la mise en œuvre d'une spécification minimale des exigences de l'adaptation de A_1 et A_2. Le contrat d'adaptation décrit de manière explicite les interactions entre les deux automates d'interface A_1 et A_2 grâce à un ensemble de *règles*. Ces règles établissent les relations entre les actions non partagées de A_1 et A_2.

Définition 5.1 (*Contrat d'adaptation*)**.** *Soient deux automates d'interface composables A_1 et A_2, une* règle d'adaptation α *est un tuple* $\langle L_1, L_2 \rangle \in (2^{\Sigma^{\mathrm{O}}_{A_1}} \times 2^{\Sigma^{\mathrm{I}}_{A_2}}) \cup (2^{\Sigma^{\mathrm{I}}_{A_1}} \times 2^{\Sigma^{\mathrm{O}}_{A_2}})$ *tels que* $(L_1 \cup L_2) \cap \mathit{Partagées}(A_1, A_2) = \emptyset$ *et si* $|L_1|$ [1] > 1 (*ou* $|L_2| > 1$) *alors* $|L_2| = 1$ (*ou* $|L_1| = 1$).

Un contrat d'adaptation $\Phi(A_1, A_2)$ *de A_1 et A_2 est un ensemble non vide de règles α_i pour* $1 \leq i \leq |\Phi(A_1, A_2)|$.

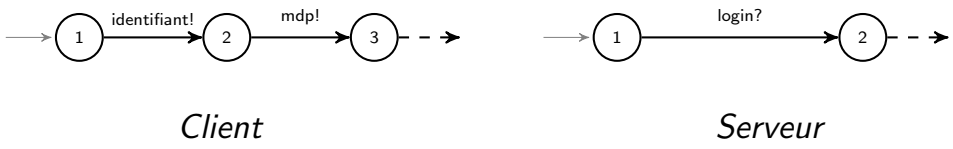


FIGURE 5.2 – Une règle d'adaptation « plusieurs-pour-une »

Nous dénotons, par $\mathit{ActIncomp}(\Phi(A_1, A_2))$, l'ensemble $\{a \in \Sigma^{\mathrm{ext}}_{A_1} \cup \Sigma^{\mathrm{ext}}_{A_2} \mid (\exists\, \alpha \in \Phi(A_1, A_2) \mid a \in \Pi_1(\alpha) \vee a \in \Pi_2(\alpha))\}$ [2]. Dans le cas où le contrat d'adaptation $\Phi(A_1, A_2)$ n'est pas défini alors l'adaptation de A_1 et A_2 ne peut pas être effectuée et leur composition est définie par le produit $A_1 \otimes A_2$. Selon la définition précédente, une règle d'adaptation autorise des correspondances « une-pour-une », « une-pour-plusieurs » et « plusieurs-pour-une » entre les actions. L'adaptation fait correspondre une ou plusieurs actions d'un automate avec une seule action de l'autre. Par exemple, un client s'authentifie en envoyant d'abord son nom d'utilisateur et ensuite un mot de passe en appelant deux opérations séparées (*identifiant* et *mdp*) alors que le serveur accepte les données en une seule opération (*login*) (cf. figure 5.2). Les correspondances « plusieurs-pour-plusieurs » ne sont pas considérées car elles peuvent être décomposées en plusieurs correspondances « une-pour-une » et « une-pour-plusieurs ».

Exemple 5.1. À titre d'exemple, nous allons considérer un système composé de deux composants logiciels : un composant client qui communique avec un serveur d'applications distant. Nous avons choisi un exemple simple pour expliquer aux lecteurs le processus d'adaptation des automates d'interface de manière simple et concise. Les automates d'interface A_C et A_S des deux composants sont montrés dans la figure 5.3.

Si l'authentification s'achève avec succès, le client envoie une requête *req* ! qui a pour paramètres son identifiant de connexion. Il envoie après, en argument de l'action *arg* !, le nom d'un

1. $| E |$ est la cardinalité d'un ensemble E.
2. Soit un tuple $\langle a, b \rangle$, $\Pi_1(\langle a, b \rangle) = a$ et $\Pi_2(\langle a, b \rangle) = b$.

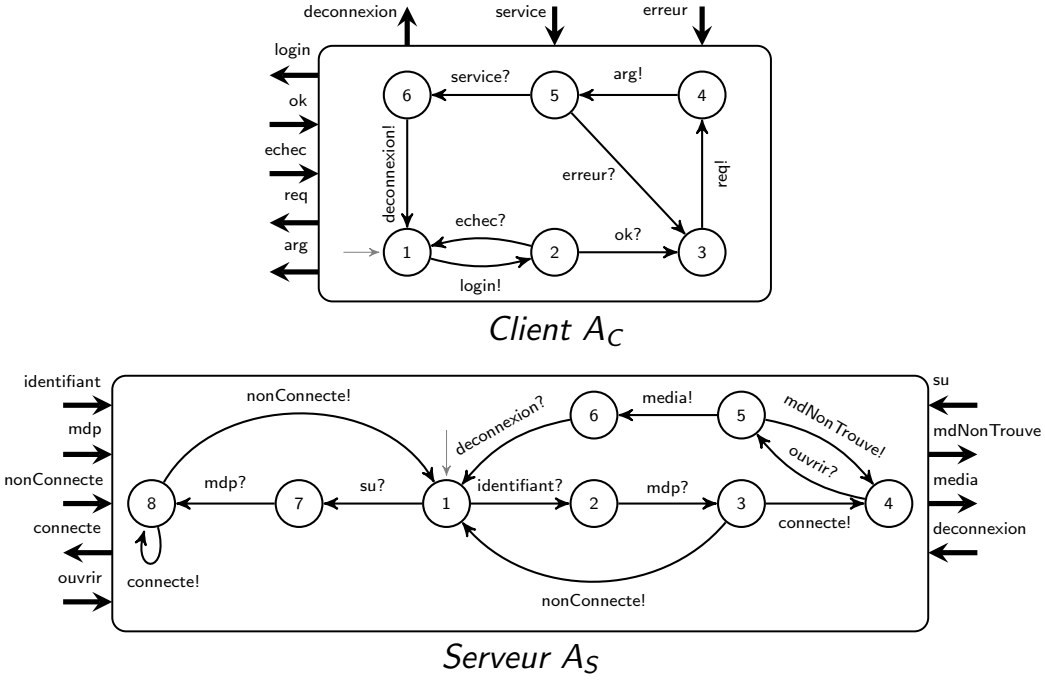


FIGURE 5.3 – Les automates d'interface d'un composant client et d'un serveur d'applications

média à ouvrir. Le serveur traite les deux actions en exécutant l'action *open* ? qui vérifie en invoquant la base de données, si le média demandé existe et si le client est autorisé à visualiser son contenu. Si le média existe dans la base de données et si le client est autorisé à l'ouvrir, le serveur renvoie le média (action de sortie *media* !) comme valeur de retour. L'action *media* ! correspond à l'action d'entrée *service* ? du client. Dans le cas où le média n'est pas trouvé, un message d'erreur (action *mdNonTrouve* !) est envoyé au client. Pour se déconnecter, le client envoie une requête *deconnexion* ! au serveur. L'action *su* ? représente l'opération fournie par le serveur qui permet d'établir la connexion d'un super-utilisateur pour ouvrir une session. L'identifiant de l'administrateur est envoyé en argument de *su*. Le serveur reçoit son mot de passe après.

Les deux automates A_C et A_S sont composables. L'ensemble des actions partagées *Partagées*(A_C, A_S) égal à $\{deconnexion\}$. Le contrat d'adaptation $\Phi(A_C, A_S)$ de A_C et A_S est défini par l'ensemble $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ tel que :

- $\alpha_1 = \langle\{login\}, \{identifiant,mdp\}\rangle$;
- $\alpha_2 = \langle\{ok\}, \{connecte\}\rangle$;
- $\alpha_3 = \langle\{echec\}, \{nonConnecte\}\rangle$;
- $\alpha_4 = \langle\{req,arg\},\{ouvrir\}\rangle$;
- $\alpha_5 = \langle\{erreur\},\{mdNonTrouve\}\rangle$;
- $\alpha_6 = \langle\{service\},\{media\}\rangle$.

Les règles du contrat d'adaptation dont les actions ont des signatures sont décrites dans le tableau 5.1. Les actions *ok*, *connecte*, *service* et *media* n'ont pas de signature car elles représentent des valeurs de retour. ■

Les règles d'un contrat d'adaptation de deux automates d'interface composables sont soumises

	A_C	A_S
α_1	$login(uid,mdp) \rightarrow (usr)$	$identifiant(id) \rightarrow ()$
		$mdp(pass) \rightarrow (u)$
α_3	$echec(msg) \rightarrow ()$	$nonConnecte(errmsg) \rightarrow ()$
α_4	$req(uid) \rightarrow ()$	$ouvrir(id,nom) \rightarrow (media)$
	$arg(media) \rightarrow (fichier)$	
α_5	$erreur(msg) \rightarrow ()$	$mdNonTrouve(errmsg) \rightarrow ()$

TABLE 5.1 – Signatures des actions dans $ActIncomp(\Phi(A_C, A_S))$

à un critère très important appelé l'*atomicité* d'exécution des actions imposées par les contraintes liées aux composants. Soient deux automates d'interface A_1 et A_2 composables et un contrat d'adaptation $\Phi(A_1, A_2)$, une règle α est dite atomique si l'adaptateur ne peut pas traiter d'autres règles α' durant son traitement. Contrairement, une règle α n'est pas atomique si l'adaptateur autorise l'exécution d'autres règles α' durant son traitement. Pour expliquer la notion d'atomicité des règles d'adaptation, nous allons présenter des exemples dans les sections suivantes.

5.2.1 Règles d'adaptation non atomiques

Soient deux composants C et D qui communiquent via un adaptateur E. Les automates d'interface des composants C et D sont présentés dans la figure 5.4.

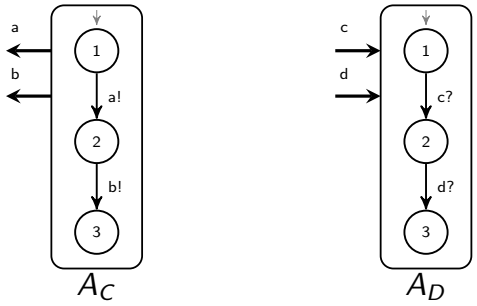


FIGURE 5.4 – Les automates d'interface A_C et A_D

Les actions de sortie $a!$ et $b!$ dans A_C correspondent respectivement aux appels des méthodes `a()` et `b()` offertes par l'adaptateur E. Les actions d'entrée $c?$ et $d?$ dans A_D correspondent respectivement aux méthodes `c()` et `d()` offertes par le composant D.

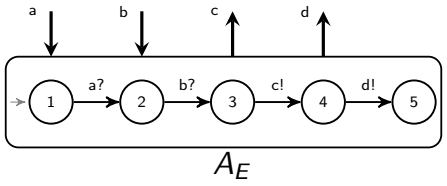


FIGURE 5.5 – L'automate d'interface A_E

Le contrat d'adaptation est défini par les règles $\langle\{a\}, \{d\}\rangle$ et $\langle\{b\}, \{c\}\rangle$. Le composant D exige que la méthode `d()` ne puisse être appelée que si la méthode `c()` soit appelée. L'adaptateur reçoit

l'appel de la méthode `a()` et celui de la méthode `b()` puis il invoque dans l'ordre la méthode `c()` qui correspond à l'appel de la méthode `b()` puis la méthode `d()` qui correspond à l'appel de la méthode `a()`.

Nous pouvons remarquer que l'appel de la méthode `a()` (action $a!$) ne puisse pas être traité par l'adaptateur directement par l'invocation de la méthode `d()` (action $d?$), les deux actions $a!$ et $d?$ sont alternées par le traitement des actions $b!$ et $c?$. Dans ce cas, nous disons que la règle $\langle\{a\},\{d\}\rangle$ n'est pas atomique car elle ne peut pas être traitée avant le traitement de la règle $\langle\{b\},\{c\}\rangle$. L'automate d'interface A_E de l'adaptateur E est présenté dans la figure 5.5. Nous remarquons que les actions de la règle $\langle\{a\},\{d\}\rangle$ sont alternées par les actions de la règle $\langle\{b\},\{c\}\rangle$.

5.2.2 Règles d'adaptation atomiques

Nous allons maintenant considérer deux autres composants P et Q qui communiquent via un adaptateur R.

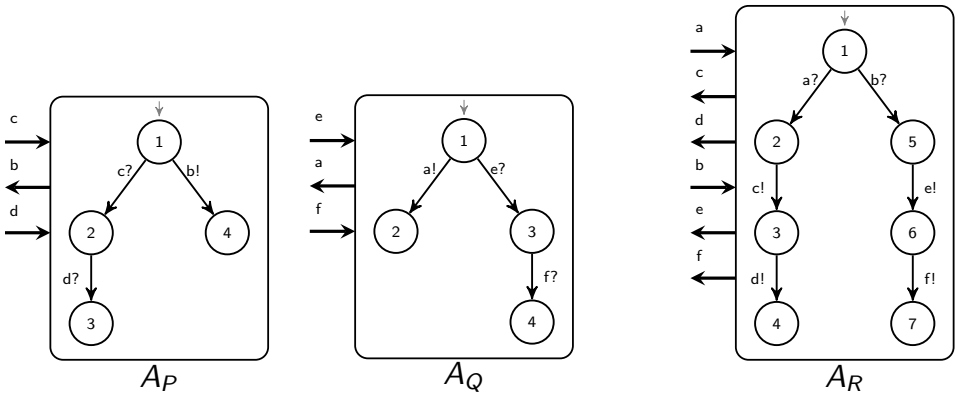


FIGURE 5.6 – Les automates d'interface A_P, A_Q et A_R

Les actions de sortie $a!$ et $b!$ respectivement dans A_Q et A_P correspondent respectivement aux appels des méthodes `a()` et `b()` offertes par l'adaptateur R. Les actions d'entrée $c?$ et $d?$ dans A_P correspondent respectivement aux méthodes `c()` et `d()` offertes par le composant P. Les actions d'entrée $e?$ et $f?$ dans A_Q correspondent respectivement aux méthodes `e()` et `f()` offertes par le composant Q.

Le contrat d'adaptation de A_P et A_Q est défini par $\Phi(A_P, A_Q) = \{\langle\{c,d\},\{a\}\rangle, \langle\{b\},\{e,f\}\rangle\}$. Le composant P exige que si l'appel de la méthode `b()` (action $b!$) soit effectué, alors les deux méthodes `c()` et `d()` (actions $c?$ et $d?$) ne peuvent pas être invoquées. Pareillement, le composant Q exige que si l'appel de la méthode `a()` (action $a!$) soit effectué, alors les deux méthodes `e()` et `f()` (actions $e?$ et $f?$) ne peuvent pas être invoquées. Les automates d'interface A_P et A_Q, présentés dans la figure 5.6, traduisent ces exigences. Si l'adaptateur E reçoit l'appel de la méthode `a()` alors, il doit invoquer les deux méthodes `c()` et `d()`. Sinon, s'il reçoit l'appel de la méthode `b()` alors, il doit invoquer les deux méthodes `e()` et `f()`.

Nous pouvons remarquer que les règles d'adaptation $\langle\{c,d\},\{a\}\rangle$ et $\langle\{b\},\{e,f\}\rangle$ sont traitées en exclusion mutuelle par l'adaptateur. Par conséquent, le traitement de la règle $\langle\{c,d\},\{a\}\rangle$ ne peut pas être alterné par le traitement de la règle $\langle\{b\},\{e,f\}\rangle$ et vice versa. Par exemple, le traitement de l'action b par l'adaptateur directement après l'appel de l'action a n'est pas autorisé. Pareillement, le traitement de l'action a directement après l'appel de l'action b n'est

pas autorisé. Dans ce cas, nous disons que les règles $\langle\{c, d\}, \{a\}\rangle$ et $\langle\{b\}, \{e, f\}\rangle$ sont des règles atomiques.

Nous notons par $\Phi^{\text{at}}(A_1, A_2) \subseteq \Phi(A_1, A_2)$, le sous ensemble des règles atomiques d'un contrat d'adaptation $\Phi(A_1, A_2)$ de deux automates d'interface A_1 et A_2.

Exemple 5.2. Nous notons que toutes les règles d'adaptation sont atomiques ($\Phi^{\text{at}}(A_C, A_S) = \Phi(A_C, A_S)$). Pour expliquer la raison pour laquelle nous avons choisi que les règles d'adaptation dans $\Phi(A_C, A_S)$ soient atomiques, nous prenons, par exemple, la règle $\langle\{$*login*$\}, \{$*identifiant*,*mdp*$\}\rangle$. Cette règle ne peut pas être alternée par n'importe quelle autre règle. Par exemple, elle ne peut pas être alternée par l'exécution de la règle $\langle\{$*ok*$\}$,$\{$*connecte*$\}\rangle$ ou $\langle\{$*echec*$\}$, $\{$*nonConnecte*$\}\rangle$, parce que l'envoi de la valeur de retour ou le traitement de l'exception ne puisse pas être effectué avant le traitement complet de l'appel *login* ! en exécutant les actions *identifiant* et *mdp*. L'atomicité des autres règles est tirée de ce principe. ■

Une règle d'adaptation atomique, dans un contrat d'adaptation $\Phi(A_1, A_2)$ de deux automates d'interface A_1 et A_2, doit être traitée de manière atomique sans l'alterner par le traitement des autres règles. Cependant, nous ne savons pas, sous quelles conditions, ses règles autorisent le traitement des actions non partagées et non reliées par le contrat d'adaptation (les actions $a \in (\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus (\mathit{Partagées}(A_1, A_2) \cup \mathit{ActIncomp}(\Phi(A_1, A_2))))$. Dans certains cas, l'appel d'une méthode provoque l'appel d'une autre méthode dans un autre composant non impliqué dans l'opération de l'adaptation. Une action d'entrée a appartenant à $\mathit{ActIncomp}(\Phi(A_1, A_2))$ peut provoquer l'exécution d'autres actions non partagées indispensables pour que l'exécution de cette action s'achève correctement. Dans la sous section suivante, nous allons donner un exemple qui explique ces intuitions.

5.2.3 Actions dépendantes

Soient deux automates d'interface A_1 et A_2 reliés par une règle d'adaptation atomique $\langle\{a\}, \{b, c\}\rangle$. L'action de sortie a ! correspond aux appels des méthodes `b()` et `c()` (actions d'entrée b ? et c ?). L'action de sortie x ! est provoquée par l'action b ? (l'exécution de la méthode `b()` provoque l'appel d'une méthode `x()`) (cf. figure 5.7). Dans ce cas, le traitement de la règle $\langle\{a\}, \{b, c\}\rangle$ peut être alternée par l'exécution de l'action x si x appartient à $(\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus (\mathit{Partagées}(A_1, A_2) \cup \mathit{ActIncomp}(\Phi(A_1, A_2))$. L'action x est dite *dépendante* de la règle $\langle\{a\}, \{b, c\}\rangle$.

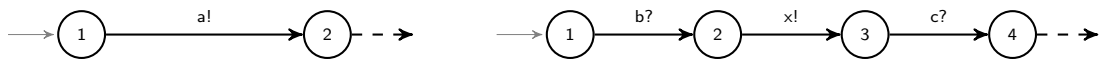


FIGURE 5.7 – L'actions x est dépendante de la règle atomique $\langle\{a\}, \{b, c\}\rangle$

Dans la composition finale entre l'adaptateur, A_1 et A_2, l'action x alterne les actions b et c. Dans le cas où l'action x n'est pas dépendante de la règle $\langle\{a\}, \{b, c\}\rangle$, l'adaptateur ne peut pas continuer le traitement de la règle après le traitement de l'action b.

Définition 5.2 (*Actions dépendantes*)**.** *Soient deux automates d'interface A_1 et A_2 reliées par un contrat d'adaptation $\Phi(A_1, A_2)$, la fonction $Dep_{\Phi^{\text{at}}(A_1,A_2)} : \Phi^{\text{at}}(A_1, A_2) \rightarrow (\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus (\mathit{Partagées}(A_1, A_2) \cup \mathit{ActIncomp}(\Phi(A_1, A_2)))$ associe pour chaque règle atomique α l'ensemble des actions $Dep_{\Phi^{\text{at}}(A_1,A_2)}(\alpha)$ non partagées, non reliées par le contrat d'adaptation, et* dépendantes *de α.*

Exemple 5.3. Nous avons $Dep_{\Phi^{\mathrm{at}}(A_C,A_S)}(\alpha) = \emptyset$ pour toute $\alpha \in \Phi^{\mathrm{at}}(A_C, A_S)$. ■

5.3 Adaptabilité sémantique

L'adaptabilité sémantique entre les actions reliées par un contrat d'adaptation de deux automates d'interface composables doit être vérifiée avant la génération de l'automate adaptateur. Les actions inadéquates doivent respecter quelques contraintes au niveau de leurs sémantiques. La notion de l'adaptabilité sémantique se base essentiellement sur la notion de la compatibilité sémantique des actions partagées.

5.3.1 Préliminaires

Soient deux automates A_1 et A_2 et un contrat d'adaptation $\Phi(A_1, A_2)$, nous rappelons que l'ensemble P_a^{i} représente l'ensemble des paramètres d'entrée d'une action a et l'ensemble P_a^{o} représente l'ensemble de ses paramètres de sortie. Pour effectuer la vérification d'adaptabilité sémantique entre A_1 et A_2, il est nécessaire de vérifier les conditions suivantes pour chaque règle $\alpha = \langle L_1, L_2\rangle \in \Phi(A_1, A_2)$:

1. $\sum_{a \in L_1} |P_a^{\mathrm{i}}| = \sum_{b \in L_2} |P_b^{\mathrm{i}}|$ et $\sum_{a \in L_1} |P_a^{\mathrm{o}}| = \sum_{b \in L_2} |P_b^{\mathrm{o}}|$;
2. Si $|L_1| = 1$ et $|L_2| \geq 1$, $L_1 = \{a\}$, $L_2 = \{b_1, ..., b_{|L_2|}\}$ et $P_a^{\mathrm{o}} = \{o_a\}$, alors il existe exactement une action $b_k \in L_2$ $(1 \leq k \leq |L_2|)$ telle que $P_{b_k}^{\mathrm{o}} = \{o_{b_k}\}$, $P_{b_l}^{\mathrm{o}} = \emptyset$ pour $1 \leq l \leq |L_2|$ et $l \neq k$ et les deux paramètres de sortie o_a et o_{b_k} satisfont la condition de sous-typage des paramètres :
   - si $L_1 \subseteq \Sigma_{A_1}^{\mathrm{O}}$, alors $D_{o_a} \subseteq D_{o_{b_k}}$;
   - sinon $D_{o_a} \supseteq D_{o_{b_k}}$.

   Le tuple θ_α représente le couple (a,b_k). Si $P_a^{\mathrm{o}} = \emptyset$, (a,b_k) n'est pas défini ;
3. La condition est analogue à la précédente avec $|L_1| \geq 1$, $|L_2| = 1$, $L_1 = \{a_1, ..., a_{|L_1|}\}$ et $L_2 = \{b\}$;
4. Dans le cas où $\bigcup_{a \in L_1} P_a^{\mathrm{i}}$ et $\bigcup_{b \in L_2} P_b^{\mathrm{i}}$ ne sont pas vides, il existe une fonction non vide $\varphi_\alpha^{\mathrm{i}} : \bigcup_{a \in L_1} P_a^{\mathrm{i}} \to \bigcup_{b \in L_2} P_b^{\mathrm{i}}$ qui associe chaque paramètre d'entrée p des actions dans L_1 avec un paramètre d'entrée q des actions dans L_2. La fonction $\varphi_\alpha^{\mathrm{i}}$ doit satisfaire la condition de sous-typage des paramètres :
   - si $L_1 \subseteq \Sigma_{A_1}^{\mathrm{O}}$, alors $D_p \subseteq D_{\varphi_\alpha^{\mathrm{i}}(p)}$ où $p \in \bigcup_{a \in L_1} P_a^{\mathrm{i}}$;
   - sinon $D_{\varphi_\alpha^{\mathrm{i}}(p)} \subseteq D_p$ où $p \in \bigcup_{a \in L_1} P_a^{\mathrm{i}}$.

La première condition établit que le nombre des paramètres d'entrée (respectivement les paramètres de sortie) des actions dans L_1 est égal au nombre des paramètres d'entrée (respectivement les paramètres de sortie) des actions dans L_2. L'intuition derrière ces conditions est d'éviter les conflits entre les pré et post-conditions causés par la présence des paramètres indépendants durant la vérification de l'adaptabilité sémantique. En effet, si le nombre des paramètres n'est pas le même, les sémantiques des actions, reliées par le contrat d'adaptation, ne peuvent jamais être compatibles dans certains cas. Par exemple, soient deux action $a \in \Sigma_{A_1}^{\mathrm{O}}$ et $b \in \Sigma_{A_2}^{\mathrm{I}}$, supposant que les signatures de a et b soient respectivement $a(x) \to (o)$ et $b(x', y') \to (o')$ tels que x, x', y', o et o' soient des entiers, $\varphi_{\langle\{a\},\{b\}\rangle}^{\mathrm{i}}(x) = x'$ et o correspond à o'. Les préconditions $Pre_{\Psi_{A_1}(a)}$ et $Pre_{\Psi_{A_2}(b)}$ de a et b sont respectivement $x = 1$ et $x' = 1 \wedge y' = 1$. Après le renommage, le

lecteur peux déduire que $x = 1 \not\Rightarrow (x = 1 \wedge y = 1)$ dans le cas où a est l'action de sortie et b est l'action d'entrée. L'action a n'est pas consciente de la présence du paramètre y' de l'action b.

Les conditions 2 et 3 établissent la relation entre les paramètres de sortie des actions dans L_1 et les paramètres de sortie de l'action b_k appartenant à L_2. Nous assumons que les autres actions dans $L_2 \setminus \{b_k\}$ n'aient pas de paramètres de sortie. La quatrième condition établit les relations entre les paramètres d'entrée des actions dans L_1 et L_2 grâce à la fonction $\varphi^{\mathrm{i}}_{\alpha}$.

5.3.2 Définitions

Avant de définir l'adaptabilité sémantique entre les actions reliées par le contrat d'adaptation de deux automates d'interface, il est indispensable que les noms des paramètres soient les mêmes dans les préconditions et post-conditions de ces actions. Le renommage des paramètres d'entrée et de sortie dans les sémantiques des actions d'une règle $\langle L_1, L_2 \rangle \in \Phi(A_1, A_2)$ est défini comme suit :

- Pour toute $a \in L_1$ et $b \in L_2$, le renommage est défini par la substitution de chaque paramètre d'entrée i de a dans $Pre_{\Psi_{A_1}(a)}$ et $Post_{\Psi_{A_1}(a)}$ par $\varphi^{\mathrm{i}}_{\alpha}(i)$ ou la substitution de chaque paramètre d'entrée i' de b dans $Pre_{\Psi_{A_2}(b)}$ et $Post_{\Psi_{A_2}(b)}$ par ${\varphi^{\mathrm{i}}_{\alpha}}^{-1}(i')$ [3].
- Si le couple $\theta_\alpha = (a, b)$ existe, le renommage est défini par la substitution du paramètre de sortie o_a dans $Post_{\Psi_{A_1}(a)}$ par o_b ou la substitution du paramètre de sortie o_b dans $Post_{\Psi_{A_2}(b)}$ par o_a.

Nous notons par $\Psi_{A_1/\alpha}(a)$ et $\Psi_{A_2/\alpha}(b)$ respectivement les sémantiques des actions a dans $\Pi_1(\alpha)$ et les actions b dans $\Pi_2(\alpha)$ après le renommage des paramètres.

Définition 5.3. *Soient deux automates d'interface composables A_1 et A_2 et un contrat d'adaptation $\Phi(A_1, A_2)$ telles que les conditions 1,2,3 et 4 établies dans la section 5.3.1 soient satisfaites, l'*adaptabilité sémantique *$SemAdap_\alpha(A_1, A_2)$ d'une règle α dans $\Phi(A_1, A_2)$ est satisfaite ssi les conditions suivantes sont valides :*

1. *si $\Pi_1(\alpha) \subseteq \Sigma^{\mathrm{O}}_{A_1}$, alors*

$$\left(\begin{array}{c} \bigwedge\limits_{a \in \Pi_1(\alpha)} Pre_{\Psi_{A_1/\alpha}(a)} \Rightarrow \bigwedge\limits_{b \in \Pi_2(\alpha)} Pre_{\Psi_{A_2/\alpha}(b)} \\ \wedge \\ \bigwedge\limits_{a \in \Pi_1(\alpha)} Post_{\Psi_{A_1/\alpha}(a)} \Leftarrow \bigwedge\limits_{b \in \Pi_2(\alpha)} Post_{\Psi_{A_2/\alpha}(b)} \end{array} \right)$$

2. *si $\Pi_1(\alpha) \subseteq \Sigma^{\mathrm{I}}_{A_1}$, alors la condition est analogue à la première en inversant les implications.*

Définition 5.4 (*Adaptabilité sémantique*)**.** *A_1 et A_2 sont* sémantiquement adaptables *conformément au contrat d'adaptation $\Phi(A_1, A_2)$ ssi $SemAdap_\alpha(A_1, A_2) \equiv$ vrai pour toute règle $\alpha \in \Phi(A_1, A_2)$.*

L'adaptabilité sémantique d'une règle $\alpha \in \Phi(A_1, A_2)$ est vérifiée de la même manière que la compatibilité sémantique (cf. définition 4.4) d'une action partagée $a \in Partagées(A_1, A_2)$ sauf que, pour l'adaptation, nous ayons plusieurs actions reliées par chaque règle d'adaptation. Par conséquent, les pré et post-conditions des actions dans les ensembles $\Pi_1(\alpha)$ et $\Pi_2(\alpha)$ doivent être reliées par des conjonctions si l'un des deux ensembles contient plus d'une action.

3. f^{-1} est la fonction réciproque de f.

Exemple 5.4. Selon le tableau 5.1, les fonctions $\varphi^{\mathrm{i}}_{\alpha_3}$ et $\varphi^{\mathrm{i}}_{\alpha_5}$ sont définies par $\{msg \mapsto errmsg\}$. La fonction $\varphi^{\mathrm{i}}_{\alpha_1}$ est définie par $\{uid \mapsto id,\ mdp \mapsto pass,\ usr \mapsto u\}$. La fonction $\varphi^{\mathrm{i}}_{\alpha_4}$ est définie par $\{uid \mapsto id,\ media \mapsto nom,\ fichier \mapsto media\}$. $\theta_{\alpha_1} = (login, mdp)$ et $\theta_{\alpha_4} = (ouvrir, arg)$. Les paramètres *uid* et *id* sont des entiers. Les paramètres *mdp* de l'action *login*, *pass*, *msg*, *errmsg*, *nom* et *media* de l'action *arg* sont tous de type chaîne de caractères. Les paramètres de sortie *media* et *fichier* respectivement des actions *ouvrir* et *arg* sont des fichiers médias. Les paramètres de sortie des actions *login* et *mdp* représentent l'objet utilisateur après l'authentification. Chacun des paramètres de sortie *fichier* et *media* respectivement des actions *arg* et *ouvrir* est une référence à une structures de données qui représentent un média. Le lecteur peut déduire aisément que les conditions préliminaires sur les règles $\alpha_1, \alpha_3, \alpha_4, \alpha_5 \in \Phi(A_C, A_S)$ sont satisfaites :

- $\forall \alpha \in \{\alpha_1, \alpha_3, \alpha_4, \alpha_5\}$, $\sum_{a \in \Pi_1(\alpha)} |P^{\mathrm{i}}_a| = \sum_{b \in \Pi_2(\alpha)} |P^{\mathrm{i}}_b|$ et $\sum_{a \in \Pi_1(\alpha)} |P^{\mathrm{o}}_a| = \sum_{b \in \Pi_2(\alpha)} |P^{\mathrm{o}}_b|$;
- les conditions 2,3 et 4 de sous-typage des paramètres sont de même satisfaites.

	Client A_C	*Serveur* A_S
α_1	$Pre_{\Psi_{A_C}(login)} \equiv uid > 0 \wedge 8 \leq mdp.length() \leq 10$ $Post_{\Psi_{A_C}(login)} \equiv usr.getId()>0$	$Pre_{\Psi_{A_S}(identifiant)} \equiv id \geq 1$ $Post_{\Psi_{A_S}(identifiant)} \equiv$ *vrai* $Pre_{\Psi_{A_S}(mdp)} \equiv 6 \leq pass.length() \leq 10$ $Post_{\Psi_{A_S}(mdp)} \equiv u.getId() \geq 1$
α_3	$Pre_{\Psi_{A_C}(echec)} \equiv msg.length() \neq 0$ $Post_{\Psi_{A_C}(echec)} \equiv$ *vrai*	$Pre_{\Psi_{A_S}(nonConnecte)} \equiv errmess.length() \neq 0$ $Post_{\Psi_{A_S}(nonConnecte)} \equiv$ *vrai*
α_4	$Pre_{\Psi_{A_C}(req)} \equiv uid > 0$ $Post_{\Psi_{A_C}(req)} \equiv$ *vrai* $Pre_{\Psi_{A_C}(arg)} \equiv$ *vrai* $Post_{\Psi_{A_C}(arg)} \equiv fichier.size() > 0$	$Pre_{\Psi_{A_S}(ouvrir)} \equiv id \geq 1$ $Post_{\Psi_{A_S}(ouvrir)} \equiv media.size() > 0$
α_5	$Pre_{\Psi_{A_C}(erreur)} \equiv msg.length() \neq 0$ $Post_{\Psi_{A_C}(erreur)} \equiv$ *vrai*	$Pre_{\Psi_{A_S}(mdNonTrouve)} \equiv errmess.length() \neq 0$ $Post_{\Psi_{A_S}(mdNonTrouve)} \equiv$ *vrai*

TABLE 5.2 – Sémantique des actions dans $ActIncomp(\Phi(A_C, A_S))$

Les pré et post-conditions des actions à adapter, respectivement dans A_C et A_S, sont données dans le tableau 5.2. Après le renommages des paramètres, il est évident que l'adaptabilité sémantique est satisfaite pour toute $\alpha \in \Phi(A_C, A_S)$ ($SemAdap_\alpha(A_C, A_S) \equiv$ *vrai*). Par exemple, les conditions de $SemAdap_{\alpha_1}(A_C, A_S)$ sont les suivantes :

- $Pre_{\Psi_{A_C/\alpha_1}(login)} \Rightarrow (Pre_{\Psi_{A_S/\alpha_1}(identifiant)} \wedge Pre_{\Psi_{A_S/\alpha_1}(mdp)})$ est satisfaite ($(id > 0 \wedge 8 \leq pass.length() \leq 10) \Rightarrow (id \geq 1 \wedge 6 \leq pass.length() \leq 10)$) ;
- $(Post_{\Psi_{A_S/\alpha_1}(identifiant)} \wedge Post_{\Psi_{A_S/\alpha_1}(mdp)}) \Rightarrow Post_{\Psi_{A_C/\alpha_1}(login)}$ est satisfaite ($usr.getId() \geq 1 \Rightarrow usr.getId() > 0$). ■

5.4 Spécification de l'adaptateur

Un automate d'interface adaptateur Ad de deux automates d'interface A_1 et A_2 conformément à un contrat d'adaptation $\Phi(A_1, A_2)$, est un automate d'interface qui leur permet d'interopérer

malgré les disparités entre leurs actions non partagées. L'adaptateur doit être composable avec A_1 et A_2 et doit respecter un ensemble de contraintes. En effet, il doit être construit en respectant (i) l'atomicité des règles d'adaptation et (ii) l'ordonnancement des événements de A_1 et A_2, en recevant toujours les actions de sortie d'un automate avant de délivrer des réponses à leurs actions d'entrée correspondantes de l'autre automate conformément à $\Phi(A_1, A_2)$.

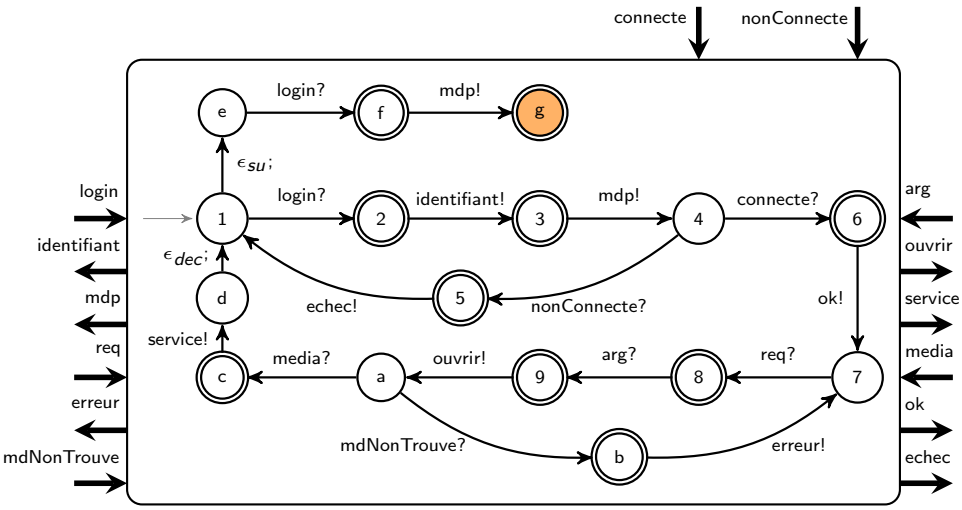


FIGURE 5.8 – Adaptateur Ad de A_C et A_S

Les concepts présentés dans cette section sont nécessaires pour comprendre l'algorithme de génération de l'automate adaptateur (cf. section 5.6). La définition formelle de l'adaptateur de deux automates d'interface composables conformément à un contrat d'adaptation est la suivante.

Définition 5.5 (*Adaptateur*)**.** *Soient deux automates d'interface A_1, A_2 composables et un contrat d'adaptation $\Phi(A_1, A_2)$, un* adaptateur *de A_1 et A_2 conformément à $\Phi(A_1, A_2)$ est un automate d'interface non vide $Ad = \langle\, S_{Ad},\, S^{\mathrm{f}}_{Ad},\, i_{Ad},\, \Sigma^{\mathrm{I}}_{Ad},\, \Sigma^{\mathrm{O}}_{Ad},\, \Sigma^{\mathrm{H}}_{Ad},\, \delta_{Ad},\, \Psi_{Ad} \rangle$ telle que :*

1. $S^{\mathrm{f}}_{Ad} \subseteq S_{Ad}$ *est l'ensemble des états dits* fermés *;*
2. $\Sigma^{\mathrm{I}}_{Ad} = \{a \mid a \in \ ActIncomp(\Phi(A_1, A_2)) \cap (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2})\}$ *;*
3. $\Sigma^{\mathrm{O}}_{Ad} = \{a \mid a \in \ ActIncomp(\Phi(A_1, A_2)) \cap (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2})\}$ *;*
4. $\Sigma^{\mathrm{H}}_{Ad} = \{\epsilon_a \mid a \in (\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus\ ActIncomp(\Phi(A_1, A_2))\}$*. Pour chaque action a qui n'appartient pas $ActIncomp(\Phi(A_1, A_2))$, l'action interne ϵ_a symbolise l'événement non opérationnel associé à a ;*
5. $\delta_{Ad} \subseteq S_{Ad} \times \Sigma_{Ad} \times S_{Ad}$ *;*
6. $Partagées(Ad, A_1) = \bigcup_{\alpha \in \Phi(A_1,A_2)} \Pi_1(\alpha)$ *et* $Partagées(Ad, A_2) = \bigcup_{\alpha \in \Phi(A_1,A_2)} \Pi_2(\alpha)$ *;*
7. *pour toute* $a \in \Sigma^{\mathrm{I}}_{Ad}$, $\Psi_{Ad}(a) = \Psi_{A_1}(a)$ *si* $a \in \Sigma^{\mathrm{O}}_{A_1}$, *sinon,* $\Psi_{Ad}(a) = \Psi_{A_2}(a)$ *;*
8. *pour toute* $a \in \Sigma^{\mathrm{O}}_{Ad}$, $\Psi_{Ad}(a) = \Psi_{A_1}(a)$ *si* $a \in \Sigma^{\mathrm{I}}_{A_1}$, *sinon,* $\Psi_{Ad}(a) = \Psi_{A_2}(a)$ *;*

L'ensemble $S^{\mathrm{f}}_{Ad} \subseteq S_{Ad}$ représente les états finaux des exécutions de Ad tel que le traitement d'une règle atomique ne soit pas encore terminé. Ces états sont marqués par l'algorithme durant le traitement des règles atomiques du contrat d'adaptation $\Phi(A_1, A_2)$. À partir d'un état fermé, soit une action d'une règle atomique α est activée ou une action ϵ_a, qui correspond à une action $a \in Dep_{\Phi^{\mathrm{at}}(A_1,A_2)}(\alpha)$ dépendante de α, est activée.

Les conditions 2 et 3 de la définition exigent que pour toute action $a \in ActIncomp(\Phi(A_1, A_2))$, si a appartient à $\Sigma^{O}_{A_1}$ ou $\Sigma^{O}_{A_2}$ alors a appartient à Σ^{I}_{Ad}, sinon si a appartient à $\Sigma^{I}_{A_1}$ ou $\Sigma^{I}_{A_2}$ alors a appartient à Σ^{O}_{Ad}. La condition 4 exige qu'une action interne non opérationnelle ϵ_a est associée pour chaque action a dans l'ensemble $(\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus ActIncomp(\Phi(A_1, A_2))$. L'algorithme de génération automatique de l'adaptateur remplace chaque transition atteignable étiquetée par une action a de l'ensemble $(\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus ActIncomp(\Phi(A_1, A_2))$ dans A_1 ou A_2 par une transition étiquetée par une action ϵ_a. Une action ϵ_a synchronise avec l'action a correspondante dans le produit final entre les deux automates et l'adaptateur.

Les conditions 7 et 8 exigent que la sémantique d'une action externe a dans Σ^{ext}_{Ad} soit la même que celle de sa correspondante dans l'un des deux automates.

Exemple 5.5. L'automate d'interface montré dans la figure 5.8 est l'adaptateur de A_C et A_S qu'on souhaite avoir (qu'on obtiendrait en appliquant l'algorithme présenté dans la section 5.6), conformément au contrat d'adaptation $\Phi(A_C, A_S)$ (cf. exemples 5.1). L'adaptateur est conçu en fonction de l'ordonnancement des événements des deux automates d'interface A_C et A_S et il reçoit les actions de sortie de $\Phi(A_C, A_S)$ avant d'émettre des actions de sortie pour leurs entrées correspondantes.

La transition (d,ϵ_{dec};,1) remplace les transitions (6,*deconnexion*!, 1) de A_C et (6,*deconnexion*?,1) de A_S dans l'adaptateur. La transition (1,ϵ_{su};,e) remplace la transition (1,*su*?,7) de A_S dans l'adaptateur. L'automate satisfait toutes les conditions de la définition 5.5. Les états {2, 3, 5, 6, 8, 9, b, c, f, g} sont les états fermés (représentés par des états doubles dans l'automate). L'exécution des actions des règles d'adaptation atomiques appartenant à $\Phi^{at}(A_C, A_S)$ passe par ces états (nous rappelons que $\Phi^{at}(A_C, A_S) = \Phi(A_C, A_S)$). À titre d'exemple, à partir de l'état g, les actions des autres règles différentes de la règle $\alpha = \langle\{login\},\{identifiant,mdp\}\rangle$ ne peuvent pas être traitées car α est atomique. Pareillement, à partir de l'état 2, l'action non partagée *su* ne peut pas alterner le traitement de la règle $\langle\{login\}, \{identifiant, mdp\}\rangle$ car $Dep_{\Phi^{at}(A_1,A_2)}(\alpha) = \emptyset$ (cf. exemple 5.3). ■

La propriété suivante peut être vérifiée aisément en se basant sur les définitions 5.5 et 4.2.

Propriété 5.1. *Soient deux automates d'interface composables A_1 et A_2, les deux propriétés suivantes sont satisfaites par l'adaptateur Ad des deux automates conformément à un contrat d'adaptation $\Phi(A_1, A_2)$:*

1. *Ad est composable avec A_1 et A_2 ;*
2. *pour toute action $a \in Partagées(Ad, A_1)$, $SemComp_a(Ad, A_1) \equiv$ vrai et pour toute action $a \in Partagées(Ad, A_2)$, $SemComp_a(Ad, A_2) \equiv$ vrai.*

Avant de définir le calcul d'adaptation, nous commençons par introduire les états bloquants dans l'adaptateur Ad de deux automates. Ces états sont les états finaux de toute exécution σ de Ad tel qu'il existe au moins une règle $\alpha \in \Phi(A_1, A_2)$ dont les actions ne sont pas toutes traitées dans σ. Ces états sont nécessaires pour définir les états illégaux dans le produit final entre l'adaptateur et les deux automates. Soit un état s, nous dénotons, par $deg^{o}_{T}(s)$, le nombre de transitions dont s est l'origine dans un ensemble de transitions T. Soient un automate d'interface A, une action $a \in \Sigma_A$ et une exécution σ de A, nous notons par $Occ_\sigma(a)$, le nombre de transitions étiquetées par l'action a sur l'exécution σ.

Définition 5.6 (*États bloquants*)**.** *Soit un adaptateur Ad de deux automates d'interface A_1 et A_2 conformément à un contrat d'adaptation $\Phi(A_1, A_2)$, l'ensemble des* états bloquants *de Ad (noté S^{b}_{Ad}) est défini par l'ensemble des états $s \in S_{Ad}$ atteignables par des exécutions σ telles que les deux conditions suivantes soient satisfaites :*

1. $deg^{\mathrm{o}}_{\delta_{Ad}}(s) = 0$;
2. *il existe* $\alpha \in \Phi(A_1, A_2)$ *dont au moins deux actions* a *et* a' *dans* $(\Pi_1(\alpha) \cup \Pi_2(\alpha))$, $Occ_\sigma(a') \neq Occ_\sigma(a)$.

Exemple 5.6. L'état "g" dans Ad est bloquant car pour toutes les exécutions qui partent de l'état initial et qui finissent par cet état, l'action *identifiant* de la règle $\langle\{login\},\{identifiant,mdp\}\rangle$ n'est pas traitée. ■

5.5 Calcul de l'adaptation

Dans cette section, nous allons présenter la définition formelle du produit synchronisé $\circledast_{Ad}$ utilisé pour calculer l'adaptation de deux automates d'interface A_1 et A_2 avec l'adaptateur Ad conformément à un contrat d'adaptation $\Phi(A_1, A_2)$.

Définition 5.7 (*Produit synchronisé* $\circledast_{Ad}$)**.** *Soient deux automates d'interface* A_1, A_2 *composables, un contrat d'adaptation* $\Phi(A_1, A_2)$, *et un adaptateur* Ad *non vide de* A_1 *et* A_2 *conforme à* $\Phi(A_1, A_2)$, *le* produit synchronisé $A_1 \circledast_{Ad} A_2$ *est défini par*

- $S_{A_1 \circledast_{Ad} A_2} = S_{A_1} \times S_{Ad} \times S_{A_2}$ *et* $i_{A_1 \circledast_{Ad} A_2} = (i_{A_1}, i_{Ad}, i_{A_2})$;
- $\Sigma^{\mathrm{I}}_{A_1 \circledast_{Ad} A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus (Partagées(A_1, A_2) \cup ActIncomp(\Phi(A_1, A_2)))$;
- $\Sigma^{\mathrm{O}}_{A_1 \circledast_{Ad} A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus (Partagées(A_1, A_2) \cup ActIncomp(\Phi(A_1, A_2)))$;
- $\Sigma^{\mathrm{H}}_{A_1 \circledast_{Ad} A_2} = (\Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2}) \cup (Partagées(A_1, A_2) \cup ActIncomp(\Phi(A_1, A_2)))$;
- $((s_1, s, s_2), a, (s'_1, s', s'_2)) \in \delta_{A_1 \circledast_{Ad} A_2}$ *si*
  - $a \in Partagées(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge (s, \epsilon_a; , s') \in \delta_{Ad}$;
  - $a \in Partagées(A_1, Ad) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2 \wedge (s, a, s') \in \delta_{Ad}$;
  - $a \in Partagées(Ad, A_2) \wedge s_1 = s'_1 \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge (s, a, s') \in \delta_{Ad}$;
  - $a \in \Sigma_{A_1} \setminus (Partagées(A_1, A_2) \cup ActIncomp(\Phi(A_1, A_2))) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge$ $s_2 = s'_2 \wedge (s, \epsilon_a; , s') \in \delta_{Ad}$;
  - $a \in \Sigma_{A_2} \setminus (Partagées(A_1, A_2) \cup ActIncomp(\Phi(A_1, A_2))) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge$ $s_1 = s'_1 \wedge (s, \epsilon_a; , s') \in \delta_{Ad}$.

Un état (s_1, s, s_2) est illégal dans le produit $A_1 \circledast_{Ad} A_2$ ssi (s_1, s_2) est illégal dans $A_1 \otimes A_2$ ou l'état s est bloquant dans l'adaptateur Ad.

Définition 5.8 (*État illégaux de l'adaptation*)**.** *L'ensemble des* états illégaux $IllAdap_{Ad}(A_1, A_2)$ *dans le produit* $A_1 \circledast_{Ad} A_2$ *est défini par* $\{(s_1, s, s_2) \in S_{A_1} \times S_{Ad} \times S_{A_2} \mid$ *l'une des conditions suivantes est satisfaite* $\}$.

- $(s_1, s_2) \in Illégaux(A_1, A_2)$;
- $s \in S^{\mathrm{b}}_{Ad}$.

Un état (s_1, s_2) est illégal dans le produit $A_1 \otimes A_2$ ssi il existe au moins une action a dans $Partagées(A_1, A_2)$ activable à partir de s_1 et elle ne l'est pas à partir de s_2 ou vice versa. Par conséquent, dans le produit final $A_1 \circledast_{Ad} A_2$, les états (s_1, s, s_2) sont de même illégaux pour toute $s \in S_{Ad}$. Si s est bloquant dans Ad alors, l'état (s_1, s, s_2) n'a pas de transitions sortantes (cf. définition 5.7). Par conséquent, il y a toujours un chemin σ entre l'état initial du produit et l'état (s_1, s, s_2) tel qu'il existe au moins une règle $\alpha \in \Phi(A_1, A_2)$ dont les actions ne sont pas toutes traitées par σ. L'ensemble des états compatibles dans $A_1 \circledast_{Ad} A_2$ est calculé conformément à la définition 3.7.

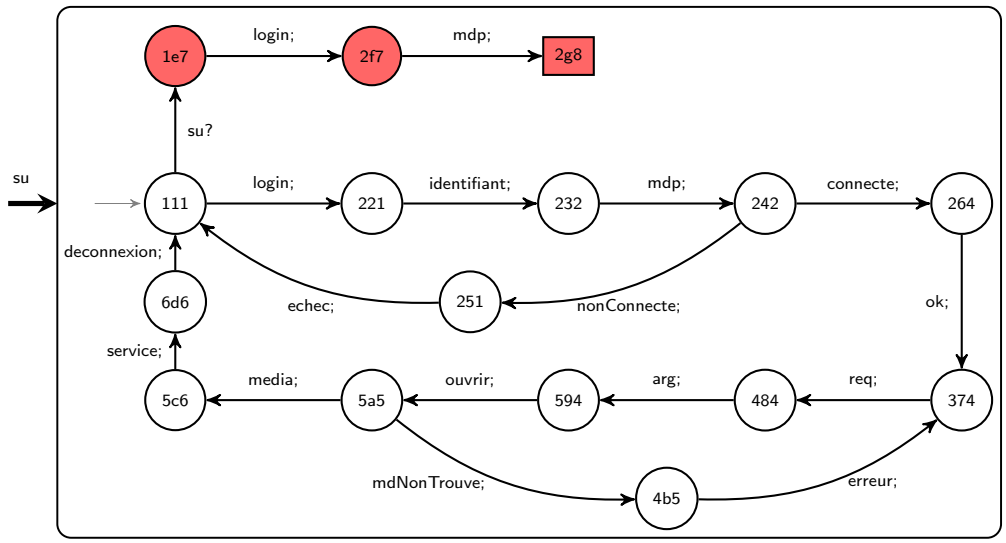


FIGURE 5.9 – Le produit $A_C \circledast_{Ad} A_S$

Définition 5.9 (*États compatibles de l'adaptation*)**.** *Un état* (s_1, s, s_2) *est* compatible *dans le produit final* $A_1 \circledast_{Ad} A_2$ *s'il n'existe pas d'état* $(s'_1, s', s'_2) \in IllAdap_{Ad}(A_1, A_2)$ *atteignable de manière autonome* (*cf. section 3.2*) *à partir de* (s_1, s, s_2)*. L'ensemble des états compatibles dans* $A_1 \circledast_{Ad} A_2$ *est noté par* $CompAdap_{Ad}(A_1, A_2)$*.*

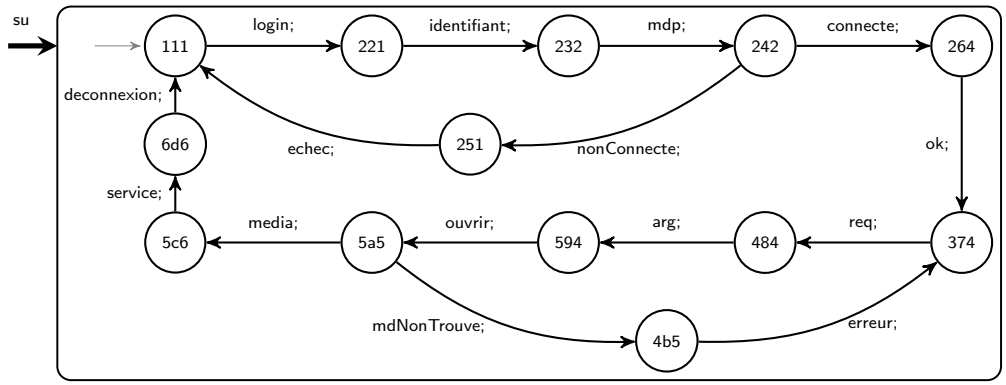


FIGURE 5.10 – L'automate d'interface $A_C \parallel_{Ad} A_S$

Définition 5.10 (*Composition finale*)**.** *La* composition finale $A_1 \parallel_{Ad} A_2$ *de* A_1 *et* A_2 *par l'adaptateur* Ad *conforme à* $\Phi(A_1, A_2)$ *est définie par la restriction de l'automate* $A_1 \circledast_{Ad} A_2$ *aux états appartenant à* $CompAdap_{Ad}(A_1, A_2)$*.*

Exemple 5.7. Reprenons les automates d'interface A_C et A_S présentés dans la figure 5.3 et l'adaptateur Ad présenté dans la figure 5.8, l'état $(2, g, 8)$ est illégal dans le produit $A_C \circledast_{Ad} A_S$ (appartient à l'ensemble $IllAdap_{Ad}(A_1, A_2)$). Conformément à la définition 5.9, l'ensemble des états incompatibles est défini par {(1,e,7),(2,f,7),(2,g,8)} (cf. figure 5.9). La composition finale $A_C \parallel_{Ad} A_S$ est non vide. ■

5.6 Algorithme de génération automatique de l'adaptateur

Dans cette section, nous présentons un algorithme permettant de construire automatiquement un adaptateur pour deux automates d'interface composables A_1 et A_2 conformément à un contrat d'adaptation $\Phi(A_1, A_2)$. L'algorithme présenté lit en parallèle A_1 et A_2 et construit au fur et à mesure l'ensemble des états et des transitions de l'adaptateur. L'algorithme est exécuté en respectant l'ordre des événements. En effet, à chaque fois qu'il traverse des transitions étiquetées par des actions de sortie appartenant à $ActIncomp(\Phi(A_1, A_2))$, il génère une ou plusieurs transitions étiquetées par leurs actions d'entrée correspondantes. L'algorithme parcourt en profondeur A_1 et A_2 en explorant alternativement leurs états et leurs transitions. Il met à jour au fur et à mesure l'ensemble S des états et l'ensemble T des transitions. Les ensembles S et T sont initialement vides.

L'algorithme permet de détecter s'il y a des incompatibilités futures, i.e. des chemins autonomes dans le produit $A_1 \circledast_{Ad} A_2$ entre l'état initial et un état illégal, qui peuvent survenir en synchronisant les actions reliées par les règles d'adaptation dans le produit final $A_1 \circledast_{Ad} A_2$. L'existence de ces chemins vont conduire à rendre leur composition finale $A_1 \parallel_{Ad} A_2$ vide. Dans ce cas, l'algorithme génère un adaptateur vide. Dans le cas contraire, il génère un adaptateur qui respecte les règles d'adaptation et assure la compatibilité de A_1 et A_2 après l'adaptation.

5.6.1 Ensemble des transitions de l'adaptateur

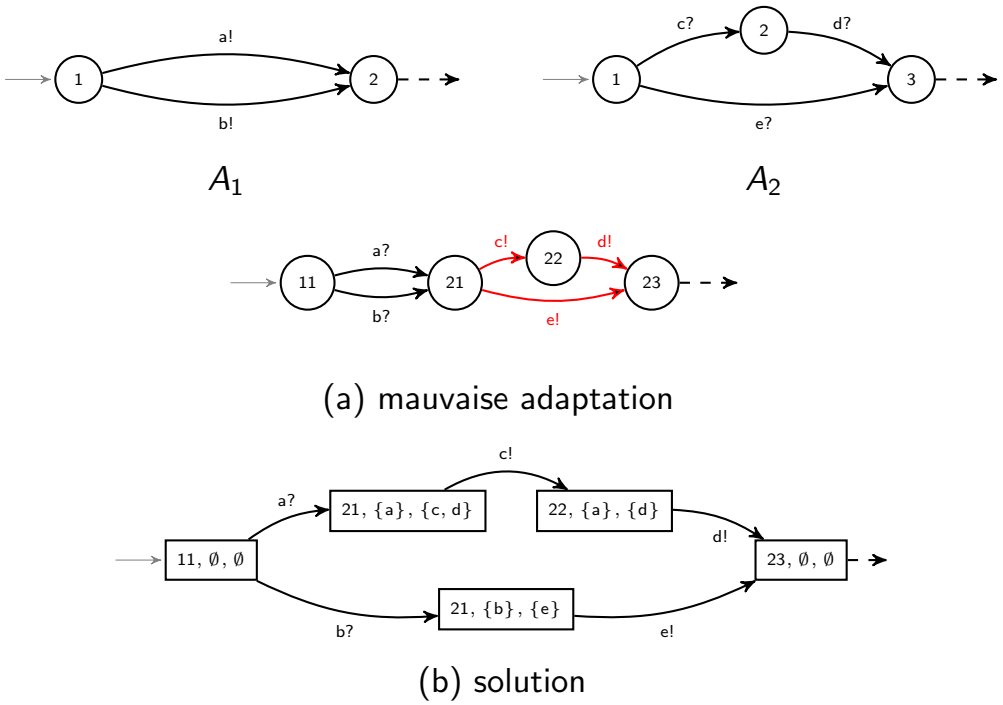


FIGURE 5.11 – Génération des transitions de l'adaptateur

Un problème crucial que nous rencontrions, c'est que nous ne puissions pas être limité au produit des états $S_{A_1} \times S_{A_1}$ de A_1 et A_2 pour construire le système des transitions de leur adaptateur. Le problème se produit lorsque, à partir d'un état s_1 dans l'un des deux automates, deux actions de sortie a et b dans $ActIncomp(\Phi(A_1, A_2))$ sont activées et elles atteignent le même

état destination s'_1, et à partir d'un état s_2 dans l'autre, leurs actions d'entrées correspondantes sont activées sur des chemins indépendants. Si nous restreignons l'ensemble des états générés à $S_{A_1} \times S_{A_1}$, nous ne pouvons pas décider quelles actions nous devons activer à partir de (s'_1, s_2) ; les actions d'entrée correspondantes à a ou celles correspondantes à b.

Pour résoudre ce problème, nous avons utilisé en complément du produit $S_{A_1} \times S_{A_2}$, une mémoire des actions de sortie consommées et celles d'entrée restantes conformément au contrat d'adaptation. À titre d'exemple, considérons une partie des deux automates d'interface A_1 et A_2, présentés dans la figure 5.11, où $a, b \in \Sigma^{\mathrm{O}}_{A_1}$ et $c, d, e \in \Sigma^{\mathrm{I}}_{A_2}$. Supposons que $\Phi(A_1, A_2) = \{\langle\{a\}, \{c, d\}\rangle, \langle\{b\}, \{e\}\rangle\}$, l'adaptateur convenable est montré dans la figure 5.11 (b).

5.6.2 Parcours en profondeur

Pour parcourir les traces de A_1 et A_2, on utilise une *pile* où les éléments sont des tuples $\langle s_1, s_2, m^{\mathrm{O}}, m^{\mathrm{I}}, aut\rangle$ tels que $s_1 \in S_{A_1}$ et $s_2 \in S_{A_2}$ sont les états courants lus par le parcours, m^{O} et m^{I} sont respectivement les mémoires des actions d'entrée et de sortie, appartenant à $ActIncomp(\Phi(A_1, A_2))$, lues durant le parcours, et *aut* est un indicateur indiquant lequel des deux automates sera parcouru durant l'itération courante après la lecture de la tête de la pile : A_1 ou A_2 ou les deux ensemble. Dans un tuple de pile τ, une action qui appartient à $ActIncomp(\Phi(A_1, A_2))$ ou non est activée à partir de $\tau.s_1$ ou $\tau.s_2$. Les transitions étiquetées par des actions $a \in (\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus ActIncomp(\Phi(A_1, A_2))$ sont remplacées par des transitions étiquetées par ϵ_a dans T. Une transition étiquetée par une action de sortie a dans $ActIncomp(\Phi(A_1, A_2))$ correspond à une transition étiquetée par $a\,?$ dans T. Une transition étiquetée par une action d'entrée b dans $ActIncomp(\Phi(A_1, A_2))$ correspond à une transition étiquetée par $b\,!$ si les actions de sortie correspondantes dans le contrat d'adaptation ont été traitées avant.

Comme nous venons de l'expliquer dans la section 5.6.1, les états de l'adaptateur sont représentés non seulement par $S_{A_1} \times S_{A_2}$, mais aussi par la mémoire des actions d'entrée et de sortie incompatibles dans $ActIncomp(\Phi(A_1, A_2))$. Les états de S sont déduits à partir des tuples de la pile. L'état qui correspond à chaque tuple de pile $\tau = \langle s_1, s_2, m^{\mathrm{O}}, m^{\mathrm{I}}, aut\rangle$ est défini par le tuple $état(\tau) = \langle \tau.s_1, \tau.s_2, \tau.m^{\mathrm{O}}, \tau.m^{\mathrm{I}}\rangle$.

Définition 5.11 (*Représentation des états bloquants*)**.** *Soit un tuple de pile τ, état(τ) est* bloquant *ssi*

1. $deg^{\mathrm{o}}_T(état(\tau)) = 0$ *;*
2. *les mémoires d'actions de τ ne sont pas vides* ($\tau.m^{\mathrm{O}} \neq \emptyset \vee \tau.m^{\mathrm{I}} \neq \emptyset$).

Nous pouvons déduire que la définition des états bloquants dans cette section est conforme à la définition 5.6. Selon la définition précédente, un état généré par l'algorithme est bloquant (i) s'il n'a pas de transition sortante (condition 1 de la définition 5.6) et (ii) si les mémoires $\tau.m^{\mathrm{O}}$ et $\tau.m^{\mathrm{I}}$ des actions d'entrée et de sortie (appartenant à $ActIncomp(\Phi(A_1, A_2))$) ne sont pas vides (condition 2 de la définition 5.6). Selon la deuxième condition de la définition, nous pouvons déduire qu'il existe au moins une règle α dans $\Phi(A_1, A_2)$ qui n'est pas complètement traitée pour chaque chemin qui termine par un état $état(\tau)$ bloquant dans l'ensemble des transitions T construit par l'algorithme.

Test de l'adaptabilité

L'algorithme permet de détecter s'il y a des chemins autonomes dans le produit $A_1 \circledast_{Ad} A_2$ entre l'état initial et un état illégal. À chaque état $\nu = \langle s_1, s_2, m^{\mathrm{O}}, m^{\mathrm{I}}\rangle$ rencontré, si (s_1, s_2)

est un état illégal dans le produit $A_1 \otimes A_2$ ou ν est bloquant, l'algorithme appelle la fonction ÉtatInitialIncompatible permettant de tester si ν est atteignable par un chemin σ dont la projection, dans le produit $A_1 \circledast_{Ad} A_2$, est autonome. Avant de présenter l'algorithme, nous introduisons un ensemble de préliminaires. Nous définissons l'opérateur $AUTpre_T(S')$ où $S' \subseteq S$ contient les états prédécesseurs des états appartenant à S' en activant uniquement des transitions étiquetées par les actions appartenant $ActIncomp(\Phi(A_1, A_2))$ (ces actions sont internes dans $A_1 \circledast_{Ad} A_2$), les actions ϵ_a où $a \in Partagées(A_1, A_2)$ (ces actions sont internes dans $A_1 \circledast_{Ad} A_2$), ou les actions ϵ_b où $b \in (\Sigma^{O}_{A_1} \cup \Sigma^{O}_{A_2}) \setminus (ActIncomp(\Phi(A_1, A_2)) \cup Partagées(A_1, A_2))$ (ces actions sont activables en sortie dans $A_1 \circledast_{Ad} A_2$). Formellement, l'opérateur $AUTpre_T$ est défini pour tout $S' \subseteq S$ par

$$AUTpre_T(S') = \{r \in S \mid (\exists (r, a, s) \in T \mid a \in ActIncomp(\Phi(A_1, A_2)) \wedge s \in S') \vee (\exists (r, \epsilon_a, s) \in T \mid a \in Partagées(A_1, A_2) \wedge s \in S') \vee (\exists (r, \epsilon_b, s) \in T \mid b \in (\Sigma^{O}_{A_1} \cup \Sigma^{O}_{A_2}) \setminus (ActIncomp(\Phi(A_1, A_2)) \cup Partagées(A_1, A_2)) \wedge s \in S')\}.$$

Fonction ÉtatInitialIncompatible$(S, T) \rightarrow$ *booléen*

Initialisation : Soit $U_0 = \{\nu \in S \mid \nu$ *est bloquant ou* $(\nu.s_1, \nu.s_2) \in Illégaux(A_1, A_2))\}$
début
  répéter
    Pour $k \geq 0$, $U_{k+1} = U_k \cup AUTpre_T(U_k)$;
  jusqu'à $U_{k+1} = U_k$;
  si *l'état initial appartient à* U_k **alors**
    retourner *vrai*;
  sinon
    retourner *faux*;
fin

Traitement de l'atomicité des règles d'adaptation

La définition suivante montre comment l'algorithme reconnait les états fermés de l'adaptateur. Un état est marqué fermé ssi l'algorithme n'a pas fini de générer les transitions étiquetées par les actions d'une règle atomique α. À partir de ces états, seules les transitions étiquetées par les actions de α sont activables ou les actions dépendante $Dep_{\Phi^{at}(A_1,A_2)}(\alpha)$ de α (cf. section 5.2).

Définition 5.12 (*Représentation des états fermés*)**.** *Soit un tuple de pile* τ*, état*(τ) *est* fermé *ssi les mémoires d'actions de* τ *ne sont pas vides et contiennent au moins une des actions d'une règle atomique* $\alpha \in \Phi^{at}(A_1, A_2)$*.*

Nous notons par $Atomique(\nu)$ la règle d'adaptation atomique α qui correspond à l'état fermé ν[4]. Nous notons par $TransPermises(\nu, A_i)$ l'ensemble des transitions de δ_{A_i} permises à partir de l'état $\nu.s_i$ pour $i \in \{1, 2\}$. Si ν est un état fermé, alors $TransPermises(\nu, A_i)$ égal à $\{(\nu.s_i, a, s'_i) \in \delta_{A_i} \mid a \in \Pi_i(\alpha) \vee a \in Dep_{\Phi^{at}(A_1,A_2)}(\alpha))\}$ où α égale à $Atomique(\nu)$. Si ν n'est pas fermé, alors $TransPermises(\nu, A_i)$ égal à $\{(\nu.s_i, a, s'_i) \in \delta_{A_i}\}$.

4. l'état parcouru en traitant la règle α.

Algorithme 3 : GénérationAdaptateur

Entrées : Deux IAs A_1 et A_2 composables et un contrat d'adaptation $\Phi(A_1, A_2) \neq \emptyset$.
Sorties : L'adaptateur *Ad*.

Initialisation : $T = S = \emptyset$ et une *pile* contenant le tuple $\langle i_{A_1}, i_{A_2}, \emptyset, \emptyset, \{A_1, A_2\}\rangle$.
début
  répéter
    $\tau \leftarrow$ *tête*(*pile*) ; *dépiler*(*pile*) ;
    si ($\tau.aut = A_1$ *ou* $\{A_1, A_2\}$) **et** *état*(τ) $\notin S$ **et** $\tau.s_1$ *n'est pas marqué* **alors**
      marquer $\tau.s_1$;
      pour chaque $(\tau.s_1, a, s'_1) \in$ *TransPermises*(*état*(τ), A_1) **faire**
        si $\exists \alpha \in \Phi(A_1, A_2) \mid a \in \Pi_1(\alpha)$ **alors**
          si $a \in \Sigma^{\mathrm{O}}_{A_1}$ **alors**
            si $\tau.m^{\mathrm{O}} \cup \{a\} \supseteq \Pi_1(\alpha)$ **alors**
              `// Toutes les actions de sortie de` α `sont traitées`
              empiler($\langle s'_1, \tau.s_2, \tau.m^{\mathrm{O}} \cup \{a\}, \Pi_2(\alpha) \cup \tau.m^{\mathrm{I}}, \{A_2\}\rangle$,*pile*) ;
            sinon
              empiler($\langle s'_1, \tau.s_2, \tau.m^{\mathrm{O}} \cup \{a\}, \tau.m^{\mathrm{I}}, \{A_1\}\rangle$,*pile*) ;
            ajouter la transition (*état*(τ), $a?$, *état*(*tête*(*pile*))) à T ;
          sinon
            si $a \in \Sigma^{\mathrm{I}}_{A_1} \cap \tau.m^{\mathrm{I}}$ **alors**
              `// a est dans la mémoire des actions d'entrée`
              si $\tau.m^{\mathrm{I}} \setminus \{a\} \neq \emptyset$ **alors**
                empiler($\langle s'_1, \tau.s_2, \tau.m^{\mathrm{O}} \setminus \Pi_2(\alpha), \tau.m^{\mathrm{I}} \setminus \{a\}, \{A_1\}\rangle$,*pile*) s'il n'y a plus d'actions d'entrée $a \in \Pi_1(\alpha)$ dans $\tau.m^{\mathrm{I}} \setminus \{a\}$;
                Sinon, *empiler*($\langle s'_1, \tau.s_2, \tau.m^{\mathrm{O}}, \tau.m^{\mathrm{I}} \setminus \{a\}, \{A_1\}\rangle$,*pile*) ;
              sinon
                `// Si la mémoire d'entrée est vide en enlevant l'action a`
                empiler($\langle s'_1, \tau.s_2, \emptyset, \emptyset, \{A_1, A_2\}\rangle$, *pile*) ;
              ajouter la transition (*état*(τ), $a!$, *état*(*tête*(*pile*))) à T ;
          sinon
            `//` $a \notin$ `ActIncomp`$(\Phi(A_1, A_2))$
            si $a \in$ *Partagées*(A_1, A_2) **et** $\exists(\tau.s_2, a, s'_2) \in \delta_{A_2}$ **alors**
              empiler($\langle s'_1, s'_2, \tau.m^{\mathrm{O}}, \tau.m^{\mathrm{I}}, \tau.aut\rangle$, *pile*) ;
            sinon
              empiler($\langle s'_1, \tau.s_2, \tau.m^{\mathrm{O}}, \tau.m^{\mathrm{I}}, \tau.aut\rangle$, *pile*) ;
            ajouter la transition (*état*(τ), $\epsilon_a;$, *état*(*tête*(*pile*))) à T ;
      si ($\tau.aut = A_2$ *ou* $\{A_1, A_2\}$) **et** *état*(τ) $\notin S$ **et** $\tau.s_2$ *n'est pas marqué* **alors**
        faire le même traitement qu'avant en l'adaptant pour toute transition $(\tau.s_2, b, s'_2)$ appartenant à *TransPermises*(*état*(τ), A_2) ;
      si *pile reste inchangée* **alors**
        si $\tau.s_1$ *n'est pas marqué* **ou** $\tau.s_2$ *n'est pas marqué* **alors**
          ré-empiler τ avec $\tau.aut = A_i$ pour l'état $\tau.s_i$ non marqué ;
        sinon
          ajouter *état*(τ) à S ;
      sinon
        ajouter *état*(τ) à S ;
  jusqu'à *pile est vide* ;
  si ÉtatInitialIncompatible(S, T) = *vrai* **alors**
    retourner *un automate d'interface vide* ;
  sinon
    retourner *un automate d'interface Ad conforme à la définition 5.5 tels que* $\delta_{Ad} = T$ *et* $S_{Ad} = S$;
fin

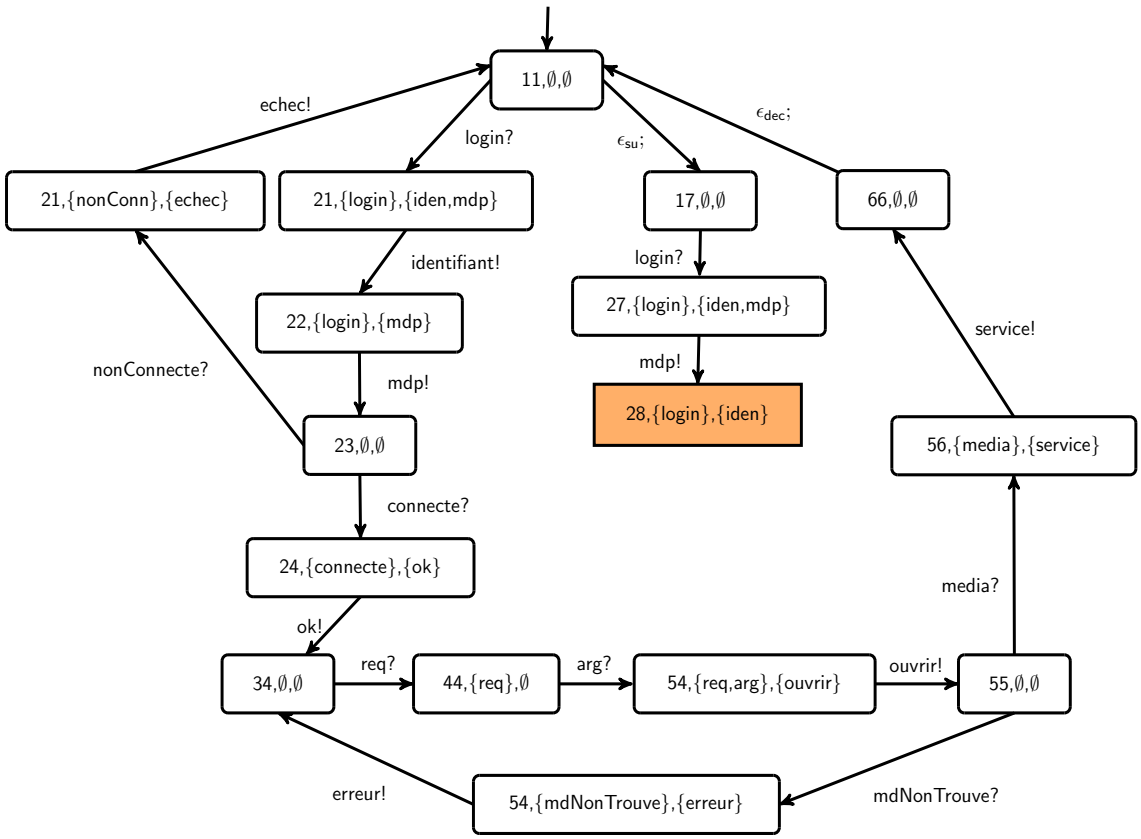


FIGURE 5.12 – Les transitions de l'adaptateur généré par l'algorithme 3 appliqué sur A_C et A_S et le contrat d'adaptation $\Phi(A_C, A_S)$

Exemple 5.8. Les transitions de l'adaptateur de A_C et A_S, généré par l'algorithme 3 conformément au contrat d'adaptation $\Phi(A_C, A_S)$, (cf. exemple 5.1) sont présentées dans la figure 5.12. L'adaptateur généré est exactement le même que celui présenté dans la figure 5.8. Selon les définitions 5.11 et 5.12, les états bloquants et fermés de l'adaptateur présenté dans la figure 5.12 sont les mêmes que ceux de l'adaptateur présenté dans la figure 5.8. ■

La complexité de l'algorithme 3, sans l'appel de la fonction ÉTATINITIALINCOMPATIBLE, est de l'ordre de $\mathbf{O}(|\delta_{A_1}| \times |\delta_{A_2}|)$. La fonction ÉTATINITIALINCOMPATIBLE permet de tester la compatibilité au pire en temps $\mathbf{O}(|\delta_{A_1}| \times |\delta_{A_2}|)$. Par conséquent, l'algorithme génère l'adaptateur pour deux automates d'interface A_1 et A_2 en temps $\mathbf{O}(|\delta_{A_1}| \times |\delta_{A_2}|)$.

5.6.3 Correction et complétude

Correction

Soient deux automates d'interface A_1 et A_2, et un contrat d'adaptation $\Phi(A_1, A_2)$, l'algorithme 3 est correct si dans le cas où il génère un automate adaptateur Ad non vide pour A_1 et A_2 conformément à $\Phi(A_1, A_2)$, alors A_1 et A_2 sont compatibles après l'adaptation par Ad. Pour prouver la correction de l'algorithme, nous devons prouver : si l'adaptateur généré par l'algorithme est non vide, alors $A_1 \parallel_{Ad} A_2$ est non vide, i.e. l'état initial du produit $A_1 \circledast_{Ad} A_2$ est compatible conformément à la définition 5.9. La preuve de la correction est composée de

plusieurs étapes. Les lemmes suivants représentent les résultats intermédiaires sur lesquels la démonstration du théorème de correction s'appuie.

Lemme 5.1. *Pour chaque transition* $((s_1, s, s_2), a, (s_1', s', s_2')) \in \delta_{A_1 \circledast_{Ad} A_2}$*, il existe une transition* $(s, a', s') \in \delta_{Ad}$ *où* $a' = a$ *si* $a \in$ *ActIncomp*$(\Phi(A_1, A_2))$*, ou* $a' = \epsilon_a$ *si* $a \notin$ *ActIncomp*$(\Phi(A_1,A_2))$.

Démonstration. La preuve de cette propriété peut être déduite à partir de la définition 5.7. Selon la définition de l'ensemble de transitions $\delta_{A_1 \circledast_{Ad} A_2}$, tout état (s_1, s, s_2) peut avoir des transitions sortantes ssi il y a des transitions sortantes dans Ad à partir de l'état s. Si $a \in$ *ActIncomp*$(\Phi(A_1, A_2))$, alors (i) il existe une transition $(s_1, a, s_1') \in \delta_{A_1}$ qui synchronise avec une transition $(s, a, s') \in \delta_{Ad}$ et $s_2' = s_2$ ou (ii) il existe une transition $(s_2, a, s_2') \in \delta_{A_2}$ qui synchronise avec une transition $(s, a, s') \in \delta_{Ad}$ et $s_1' = s_1$. Si $a \in$ *Partagées*(A_1, A_2), alors il existe trois transitions $(s_1, a, s_1') \in \delta_{A_1}$, $(s_2, a, s_2') \in \delta_{A_2}$ et $(s, \epsilon_a, s') \in \delta_{Ad}$ qui synchronisent. Pour le reste des actions non partagées et qui n'appartiennent pas à *ActIncomp*$(\Phi(A_1, A_2))$, (i) il existe une transition $(s_1, a, s_1') \in \delta_{A_1}$ qui synchronise avec une transition $(s, \epsilon_a, s') \in \delta_{Ad}$ et $s_2' = s_2$ ou (ii) il existe une transition $(s_2, a, s_2') \in \delta_{A_2}$ qui synchronise avec une transition $(s, \epsilon_a, s') \in \delta_{Ad}$ et $s_1' = s_1$. □

Lemme 5.2. *Pour chaque chemin fini* $\sigma = ia_1 s_1 a_2 s_2 ... a_n s_n$ *de taille* $n \geq 1$ *de l'automate* $A_1 \circledast_{Ad} A_2$ *tel que* $i = i_{A_1 \circledast_{Ad} A_2}$*, alors il existe un chemin* $\rho = ja_1' t_1 a_2' t_2 ... a_n' t_n$ *de la même taille que* σ *dans* Ad *tel que* $j = i_{Ad}$*,* t_k *est la projection de* s_k *dans* Ad*, et* $a_k' = a_k$ *si* $a_k \in$ *ActIncomp*$(\Phi(A_1, A_2))$*, ou* $a_k' = \epsilon_{a_k}$ *si* $a_k \notin$ *ActIncomp*$(\Phi(A_1, A_2))$ *pour* $k \in \{1, ..., n\}$.

Démonstration. La preuve est triviale en se basant sur le lemme 5.1. □

L'algorithme suppose que tout état tuple ν (généré par l'algorithme) tel que $(\nu.s_1, \nu.s_2) \in$ *Illégaux*(A_1, A_2), alors ν correspond à un seul état $(s_1, \nu, s_2) \in$ *IllAdap*$_{Ad}(A_1, A_2)$ tel que $\nu.s_1 = s_1$ et $\nu.s_2 = s_2$. Pour garantir cette supposition, il faut prouver que, en se basant sur la définition du produit $A_1 \circledast_{Ad} A_2$ (cf. définition 5.7), pour chaque état $(s_1, \nu, s_2) \in S_{A_1 \circledast_{Ad} A_2}$, nous avons $\nu.s_1 = s_1$ et $\nu.s_2 = s_2$

Lemme 5.3. *Pour tout état tuple* $\nu \in S_{Ad}$ *généré par l'algorithme 3, si* $(\nu.s_1, \nu.s_2) \in$ *Illégaux*(A_1, A_2)*, alors* ν *correspond à un état unique* $(s_1, \nu, s_2) \in$ *IllAdap*$_{Ad}(A_1, A_2)$ *tel que* $\nu.s_1 = s_1$ *et* $\nu.s_2 = s_2$.

Démonstration. Pour prouver ce lemme, il suffit de prouver que tout état $(s_1, \nu, s_2) \in S_{A_1 \circledast_{Ad} A_2}$, nous avons $\nu.s_1 = s_1$ et $\nu.s_2 = s_2$. Nous devons vérifier que, pour chaque chemin $\sigma = ia_1 s_1 a_2 s_2 ... a_n s_n$ dans $A_1 \circledast_{Ad} A_2$ tel que $i = i_{A_1 \circledast_{Ad} A_2}$, dans chaque état $s_i = (s_i^1, \nu_i, s_i^2)$ atteignable par σ, nous avons $\nu_i.s_1 = s_i^1$ et $\nu_i.s_2 = s_i^2$.

La preuve peut être itérée par induction. Dans l'état initial $i = (i_{A_1}, j, i_{A_2})$ de $A_1 \circledast_{Ad} A_2$, j correspond au tuple $\langle i_{A_1}, i_{A_2}, \emptyset, \emptyset \rangle$. Nous supposons que, pour l'état $s_k = (s_k^1, \nu_k, s_k^2)$ d'ordre k atteignable dans σ, $\nu_k.s_1$ égal à s_k^1 et $\nu_k.s_2$ égal à s_k^2. En se basant sur la définition 5.7, nous pouvons prouver que l'hypothèse est valide pour $s_{k+1} = (s_{k+1}^1, \nu_{k+1}, s_{k+1}^2)$ d'ordre $k+1$ atteignable dans σ. □

Théorème 5.4. *Soient deux automates d'interface* A_1*,* A_2 *et un contrat d'adaptation* $\Phi(A_1, A_2) \neq \emptyset$*, si l'adaptateur* Ad *généré par l'algorithme 3 appliqué sur* A_1*,* A_2 *et* $\Phi(A_1, A_2)$ *est non vide, alors* $A_1 \parallel_{Ad} A_2$ *est non vide.*

Démonstration. Prouver que $A_1 \parallel_{Ad} A_2$ est non vide si Ad n'est pas vide consiste à prouver que l'état initial du produit $A_1 \circledast_{Ad} A_2$ est compatible. Pour vérifier si l'état initial $i = i_{A_1 \circledast_{Ad} A_2}$ est compatible il suffit de prouver que tout chemin $\sigma = ia_1 s_1 a_2 s_2 ... a_n s_n$, qui part de l'état initial et qui termine par un état illégal $s_n \in \mathit{IllAdap}_{Ad}(A_1, A_2)$, ne soit pas autonome (il existe au moins une action $a_k \in \Sigma^{\mathrm{I}}_{A_1 \circledast_{Ad} A_2}$ pour $k \in \{1, ..., n\}$).

Selon le lemme 5.2, il existe un chemin $\rho = ja'_1 t_1 a'_2 t_2 ... a'_n t_n$ dans Ad qui a la même taille que σ tel que $j = i_{Ad}$, t_k est la projection de s_k dans Ad, et $a'_k = a_k$ si $a_k \in \mathit{ActIncomp}(\Phi(A_1, A_2))$ ou $a'_k = \epsilon_{a_k}$ si $a_k \notin \mathit{ActIncomp}(\Phi(A_1, A_2))$ pour $k \in \{1, ..., n\}$. Si $s_n \in \mathit{IllAdap}_{Ad}(A_1, A_2)$, alors $t_n \in S^{\mathrm{b}}_{Ad}$ ou t_n correspond à un état tuple ν tel que $(\nu.s_1, \nu.s_2) \in \mathit{Illégaux}(A_1, A_2)$ (résultat du lemme 5.3 et la définition 5.8). Chaque chemin, qui part de l'état initial j de Ad et qui termine par l'état t_n, est exploré par l'algorithme en appelant la fonction ÉtatInitialIncompatible (instructions 39 jusqu'à 42 dans l'algorithme). La fonction fait un parcours en arrière à partir de l'état bloquant t_n qui s'arrête, sur un chemin, dès qu'une transition étiquetée par une action ϵ_{a_k}, où $a_k \in \Sigma^{\mathrm{I}}_{A_1 \circledast_{Ad} A_2}$, est atteinte. Si l'algorithme retourne un adaptateur non vide, alors le chemin ρ qui correspond à σ contient une transition $(t_{k-1}, \epsilon_{a_k}, t_k)$ où $a_k \in \Sigma^{\mathrm{I}}_{A_1 \circledast_{Ad} A_2}$. Par conséquent, il existe une transition (s_{k-1}, a_k, s_k) étiquetée par une action d'entrée a_k sur le chemin σ. □

Complétude

Les propriétés de complétude sont indispensables pour que le composant adaptateur remplisse son rôle. Si l'un des deux composants demande un service, l'adaptateur reçoit la requête et il invoque toutes les opérations qui correspondent au service demandé dans l'autre composant. Pareillement, si une règle doit être exécutée de manière atomique, l'adaptateur n'interfère pas son traitement par le traitement d'autres règles.

Propriété 5.2. *Si l'automate adaptateur Ad généré n'est pas vide, l'algorithme 3 est complet ssi les trois conditions suivantes sont satisfaites :*

1. *l'algorithme termine ;*
2. *si l'automate adaptateur généré est vide, alors il n'existe aucun adaptateur qui assure la compatibilité de A_1 et A_2 conformément à $\Phi(A_1, A_2)$;*
3. *pour tout chemin σ de Ad qui ne termine pas par un état tuple ν tel que $\nu \in S^{\mathrm{b}}_{Ad}$ ou $(\nu.s_1, \nu.s_2) \in \mathit{Illégaux}(A_1, A_2)$, les deux propriétés suivantes sont satisfaites :*
   - *s'il existe au moins une action a d'une règle $\alpha \in \Phi(A_1, A_2)$ tel que a soit l'étiquette de n transitions dans σ, alors chaque action a' de la règle telle que $a' \neq a$ soit l'étiquette de n transitions dans σ,*
   - *s'il existe $\alpha \in \Phi(A_1, A_2)$ telle que les actions de sortie (activables en entrée dans Ad) de α soient activable dans σ alors elles sont suivies par leurs actions d'entrée correspondantes (activables en sortie dans Ad) ;*
4. *pour chaque état fermé s de l'adaptateur, l'algorithme n'autorise pas la génération des transitions étiquetées par des actions appartenant aux règles différentes de la règle $\mathit{Atomique}(s)$ ou une action non partagée appartenant à $\mathit{Dep}_{\Phi^{\mathrm{at}}(A_1,A_2)}(\mathit{Atomique}(s))$.*

Démonstration. Notre algorithme proposé assure la validité des ses propriétés si l'adaptateur généré n'est pas vide. Le marquage des états (instructions 5, 6, 35 et 37) assure le traitement d'un état uniquement une seule fois sur un chemin généré, ce qui assure la terminaison de l'algorithme (condition 1). Les blocs d'instructions entre les lignes 10 et 24 assure la validité de la condition 3. La fonction $\mathit{TransPermises}(\nu, A_i)$, pour $i \in \{1, 2\}$ et ν est un état tuple fermé,

restreint l'ensemble des transitions activables à partir de l'état $\nu.s_i$ de telle sorte que la fermeture de ν et l'atomicité de $Atomique(\nu)$ soient respectées (condition 4). La validité de la condition 2 est triviale car l'insatisfaction de l'une des condition 1, 3 ou 4 implique la non-conformité de l'adaptateur généré au contrat d'adaptation $\Phi(A_1, A_2)$. □

5.7 Synthèse

L'approche proposée dans ce chapitre est constituée d'un ensemble d'outils formels permettant de générer automatiquement un adaptateur des automates d'interface de deux composants incompatibles afin de garantir leur interopérabilité. Cette approche d'adaptation est basée sur les interfaces des composants et traite les trois niveaux d'interopérabilité signature, sémantique et protocole. L'originalité de ce travail est l'exploitation de l'approche optimiste des automates d'interface dans le cadre de la spécification et la génération automatique des adaptateurs.

Une première étape de notre approche consiste à considérer un contrat d'adaptation entre les noms des opérations incompatibles des deux composants, qui permet de définir des correspondances « une-pour-une », « une-pour-plusieurs » et « plusieurs-pour-une ». Cette fonction représente la spécification abstraite minimale de l'adaptateur. Ensuite, nous avons défini un ensemble de conditions pour vérifier l'adaptabilité sémantique des composants en se basant sur les automates d'interfaces et le contrat d'adaptation. Puis, nous avons défini formellement les conditions qui doivent être satisfaites par l'adaptateur et les contraintes selon lesquelles la composition de l'adaptateur avec deux automates d'interfaces est calculée. Enfin, nous avons proposé un algorithme qui permet de générer un adaptateur, pour deux automates d'interface composables, conformément à un contrat d'adaptation permettant de garantir leur compatibilité.

Chapitre 6

Automates d'interface sémantiques

Dans ce chapitre, nous augmentons le pouvoir d'expression des automates d'interface pour la vérification, d'une part, de l'interopérabilité des composants qui communiquent par des variables, et d'autre part, la sûreté des systèmes composites après le processus d'assemblage. Le nouveau formalisme des automates d'interface sémantiques (*semantical interface automata* SIAs) est doté d'un niveau sémantique d'interopérabilité plus riche que le formalisme présenté dans le chapitre 4. En effet, la description de la sémantique des actions dans ce formalisme n'est pas basée uniquement sur les paramètres des opérations mais aussi sur l'utilisation des *variables*. Ces variables sont partagées entre la spécification d'un composant et celle de son environnement et permettent d'exhiber plus d'informations sur l'interaction d'un composant avec son environnement et ainsi exploiter ces informations pour vérifier de l'interopérabilité.

En outre, nous élaborons des méthodes formelles pour spécifier et vérifier les propriétés de sûreté grâce pouvoir d'expression des automates d'interface sémantiques. Les propriétés de sûreté sont typiquement des *invariants* définis sur les variables partagées. Les propriétés d'invariance sont évaluées à tous les états d'une représentation en système de transitions étiquetées (*Labeled Transition Systems* LTS) des automates d'interface sémantiques. En particulier, nous étudions la préservation des invariants par la composition et le raffinement.

Dans la section 6.1, nous présentons la définition formelle des automates d'interface sémantiques. Dans la section 6.2, nous présentons comment la compatibilité entre les automates d'interface sémantiques est effectuée en exploitant la sémantique décrites en termes des paramètres des opérations et les variables. Dans la section 6.3, nous adaptons l'approche de simulation alternée, utilisée pour raffiner les automates d'interface sémantiques, pour supporter l'utilisation des variables. Dans la section 6.4, nous expliquons comment un automate d'interface sémantique est traduit par un système de transitions étiquetées (Représentations LTS des SIAs) et comment les invariants sont préservés par la composition et le raffinement. Dans la section 6.5, nous montrerons comment les automates d'interface sémantiques peuvent être utilisés pour générer une partie du code des composants EJB. L'approche des automates d'interface sémantiques est appliquée sur l'exemple présenté dans le chapitre 4.

6.1 Définitions

En général, spécifier un système nécessite souvent de refléter son comportement en termes d'actions et réactions de ses sous-composants communicants. Plus précisément, il est important d'avoir, à un moment donné, une description explicite de l'état du système en cours du temps. La solution commune pour la spécification de comportement évolutif d'un système est l'utilisation de

variables modifiables par ses événements. Dans les systèmes à base de composants, les interactions d'un composant avec son environnement sont souvent décrites par les protocoles qui décrivent uniquement l'ordonnancement temporel de ses entrées et sorties. Pour décrire de plus près l'état d'un système, des variables doivent être définies aux niveaux des spécifications contractuelles des interfaces des composants.

Les automates d'interface sémantiques unifient l'utilisation des variables et des paramètres des opérations pour décrire la sémantique des actions. Les pré et post-conditions sont définies sur les paramètres des actions et les variables. Les variables sont manipulées systématiquement par un ou plusieurs automates d'interface sémantiques.

Nous supposons que tous les préliminaires sur les signatures des actions présentés au début de la section 4.1 seront aussi utilisés dans la suite de ce chapitre. Soit un ensemble de composants $\mathcal{C}$, un automate d'interface sémantique A_c d'un composant $c \in \mathcal{C}$ est défini en relation avec les autres composants $\mathcal{C} \setminus \{c\}$. Cette relation est basée sur l'utilisation d'un ensemble de variables $V_{\mathcal{C}}$ partagées. De même, chaque automate d'interface sémantique peut agir sur un ensemble de variables locales. Une variable v est associée à un domaine D_v comme les paramètres des opérations (cf. section 4.1). Une *valuation* d'un ensemble de variables ou de paramètres V est une fonction

$$\phi : V \rightarrow \bigcup_{v_i \in V} D_{v_i}$$

qui associe à chaque $v_i \in V$ une valeur dans D_{v_i}. Nous notons par $\phi\langle V' \rangle$ la restriction de ϕ à l'ensemble $V' \subseteq V$. L'ensemble $\mathcal{I}_V$ est l'ensemble de toutes les valuations possibles ϕ de V. Soit un ensemble de variables V, nous rappelons que $\mathit{Preds}(V)$, l'ensemble des prédicats de premier ordre dont les variables libres appartiennent à V. Soit un prédicat $p \in \mathit{Preds}(V)$, nous notons par p' le prédicat obtenu en substituant la variable v par v' pour tout $v \in V$. L'ensemble $\mathit{Preds}'(V)$ égal à $\{p' \mid p \in \mathit{Preds}(V)\}$.

Définition 6.1 (*Automates d'Interface Sémantiques*)**.** *Soit un composant $c \in \mathcal{C}$, un* automate d'interface sémantique *A_c du composant c est un tuple $A_c = \langle S_{A_c}, i_{A_c}, \Sigma^{\mathrm{I}}_{A_c}, \Sigma^{\mathrm{O}}_{A_c}, \Sigma^{\mathrm{H}}_{A_c}, \delta_{A_c}, L_{A_c}, V_{A_c}, \mathit{Init}_{A_c}, \Psi_{A_c}, \mathit{Chg}_{A_c}, \Omega_{A_c} \rangle$ tel que :*

- *S_{A_c} est ensemble fini d'états ;*
- *$i_{A_c} \in S_{A_c}$ est l'état initial ;*
- *$\Sigma^{\mathrm{I}}_{A_c}, \Sigma^{\mathrm{O}}_{A_c}$ et $\Sigma^{\mathrm{H}}_{A_c}$ sont respectivement les ensembles des noms des actions d'entrée, de sortie et internes ;*
- *$\delta_{A_c} \subseteq S_{A_c} \times \Sigma_{A_c} \times S_{A_c}$ est l'ensemble des transitions entre les états ;*
- *Un ensemble L_{A_c} de variables locales ;*
- *Un sous ensemble V_{A_c} de $V_{\mathcal{C}}$ de variables partagées. L'ensemble $LV_{A_c} = V_{A_c} \cup L_{A_c}$ représente l'ensemble de toutes les variables de A_c ;*
- *Init_{A_c} est la valuation initiale des variables LV_{A_c} ;*
- *Ψ_{A_c} est une fonction qui associe pour chaque action $a \in \Sigma_{A_c}$ un tuple $\langle \mathit{Pre}_{\Psi_{A_c}(a)}, \mathit{Post}_{\Psi_{A_c}(a)} \rangle$ tel que $\mathit{Pre}_{\Psi_{A_c}(a)} \in \mathit{Preds}(V_{A_c} \cup P^{\mathrm{i}}_a)$ et $\mathit{Post}_{\Psi_{A_c}(a)} \in \mathit{Preds}(V_{A_c} \cup P^{\mathrm{i}}_a \cup P^{\mathrm{o}}_a)$;*
- *$\mathit{Chg}_{A_c} : \Sigma_{A_c} \rightarrow 2^{LV_{A_c}}$ est une fonction qui associe a chaque action $a \in \Sigma_{A_c}$, l'ensemble des variables $\mathit{Chg}_{A_c}(a) \subseteq LV_{A_c}$ qui sont modifiables par a ;*
- *Ω_{A_c} est une fonction qui associe pour chaque action a un prédicat de transition noté*

$$\Omega_{A_c}(a) = \bigvee_{k \geq 1} (\mathit{grd}_k \wedge \mathit{cmd}_k \wedge \mathit{Inchg}_k)$$

tels que

- $grd_k \in Preds(LV_{A_c})$ *pour* $k \geq 1$ *un prédicat qui représente une des* gardes *de l'action* a *;*
- $cmd_k \in Preds'(V_k)$*, où* $V_k \subseteq Chg_{A_c}(a)$*, est un prédicat* commande *défini en fonction des variables* primées v' *qui représentent les variables* $v \in V_k$ *après l'exécution de l'action* a *si la garde* grd_k *est satisfaite ;*
- $Inchg_k = Inchangé(LV_{A_c} \setminus V_k)$ *est un prédicat défini sur le reste des variables dans* $LV_{A_c} \setminus V_k$ *qui ne changent pas de valeurs après l'exécution de a. Le prédicat Inchangé*(V)*, pour un ensemble de variable quelconque V, est défini par*

$$\bigwedge_{v \in V} v' = v.$$

Nous exigeons, comme les chapitres 3 et 4 précédents, que les automates d'interface sémantiques sont aussi déterministes.

Nous assumons que le changement de variables est provoqué par les prédicats de transition des actions de sortie ou internes parce qu'elles sont localement contrôlées par l'automate. Les prédicats de transition des actions d'entrée décrivent comment les variables dans $Chg_{A_c}(a)$ sont mises à jour dans le cas où l'environnement appèle l'action.

Selon la définition 6.1, les sémantiques (pré et post-conditions) des actions sont définies uniquement sur les paramètres et les variables partagées et non sur les variables locales. Les pré et post-conditions sont utilisées pour vérifier la compatibilité entre deux automates d'interface sémantiques donc, elle doivent être définies uniquement sur ce qui est partagé entre eux. La présence des variables locales peut poser problème car les variables locales d'un automate ne sont pas connues aux autres.

De plus, la précondition d'une action a n'est pas suffisante pour définir la garde complète d'une action parce qu'elle est décrite uniquement en fonction des paramètres d'entrée et les variables partagées. Le prédicat de transition $\Omega_{A_c}(a)$ d'une action a peut imposer des conditions sur les variables locales (absentes dans la précondition) avant de changer les variables. Les gardes grd_k de $\Omega_{A_c}(a)$ sont utilisées pour établir ces conditions.

Définition 6.2 (*Satisfaction des prédicats de transitions*)**.** *Soit un automate d'interface sémantiques A, pour toute transition* $(s, a, s') \in \delta_A$*, uniquement les valuations* $\phi \in \mathcal{I}_{LV_A}$ *et* $\phi' \in \mathcal{I}_{LV_A}$*, qui satisfont les deux conditions suivantes, sont autorisées aux états* s *et* s' *:*

1. $(\Omega_A(a) \wedge \bigwedge_{v \in LV_A} v = \phi(v)) \equiv$ *vrai ;*
2. $(\Omega_A(a) \wedge \bigwedge_{v \in LV_A} v' = \phi'(v)) \equiv$ *vrai.*

Soit un automate d'interface sémantique A, nous notons par $\mathcal{I}^s_{LV_A} \subseteq \mathcal{I}_{LV_A}$ *l'ensemble des valuations des variables dans* LV_A *autorisées à l'état* $s \in S_A$*.*

Soit une action $a \in \Sigma_A$, nous notons par $\mathcal{E}(\phi, \Omega_A(a)) \in \mathcal{I}_{LV_A}$ la valuation des variables LV_A après l'exécution de a telle que ϕ est une valuation qui satisfait la condition 1 de la définition 6.2. La valuation $\mathcal{E}(\phi, \Omega_A(a))$ satisfait la condition 2 de la même définition.

Exemple 6.1. Nous reprenons l'exemple 4.1 présenté dans le chapitre 4. Nous allons considérer les trois composants Client, Services et LocationRessource. L'ensemble $\mathcal{C}$ des composants est donc défini par {Client, Services, LocationRessource}.

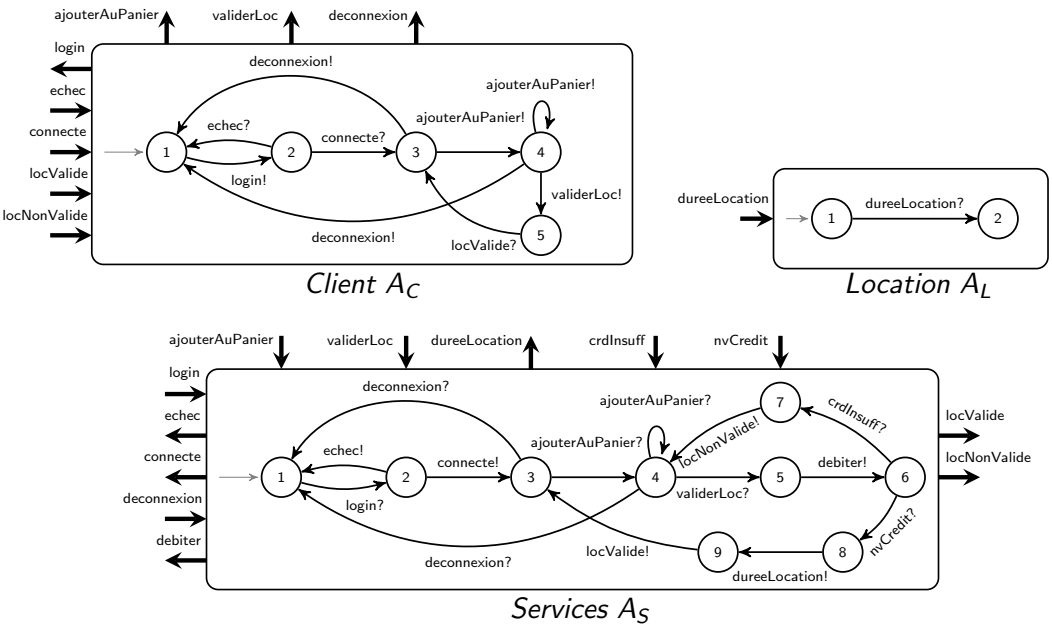


FIGURE 6.1 – Automates d'interface sémantiques A_C et A_S des composants Client et Services de l'exemple 4.1

Les automates d'interface sémantiques de ces trois composants sont présentés dans la figure 6.1 (les informations sémantiques ne sont pas montrées sur les automates). L'ensemble des variables partagées $V_{\mathcal{C}}$ est défini par {*sess*,*panier*}. La variable *sess* indique le statut de la session client. La variable *panier* indique le statut du panier virtuel. L'ensemble L_{A_C} est vide, l'ensemble L_{A_S} égal à {*tents*} et l'ensemble L_{A_L} est vide. La variable *tents* représente le nombre de tentatives de connexion autorisées par le serveur. Nous assumons que $V_{A_C} = \{sess, panier\}$, $V_{A_S} = \{sess, panier\}$ et $V_{A_L} = \{panier\}$. L'automate A_C partage avec A_S les variables *sess* et *panier*. L'automate V_{A_L} partage avec les automates A_C et A_S uniquement la variable *panier*. Le tableau 6.1 présente les sémantiques des actions partagées dans $Partagées(A_C,A_S)$ et dans $Partagées(A_S,A_L)$.

Le domaine D_{tents} égale à $\mathbb{N}$. Le domaine D_{sess} est défini par {*active*,*inactive*}. Le domaine D_{panier} est défini par {*vide*,*nonvide*}. Nous assumons que le composant Services autorise au plus deux tentatives de connexion. L'action *connecte* active la session du client. L'action *deconnexion* désactive la session du client. L'action *validerLoc* initie la location. L'action *dureeLocation* valide complètement une location, en cas où le crédit du client est suffisant, en lui affectant une durée de location par défaut et en vidant le panier ($panier = vide$). Nous assumons que $\mathit{Init}_{A_C} = \{sess \mapsto inactive, panier \mapsto vide\}$, $\mathit{Init}_{A_S} = \{sess \mapsto inactive, tents \mapsto 2, panier \mapsto vide\}$ et $\mathit{Init}_{A_L} = \{panier \mapsto vide\}$. Nous assumons que les conditions suivantes soient valides :

- $Chg_{A_C}(login) = \emptyset$ et $Chg_{A_S}(login) = \{tents\}$,
- $Chg_{A_C}(connecte) = \{sess\}$ et $Chg_{A_S}(connecte) = \{sess\}$,
- $Chg_{A_C}(deconnexion) = \{sess\}$ et $Chg_{A_S}(deconnexion) = \{tents, sess\}$,
- $Chg_{A_C}(ajouterAuPanier) = \{panier\}$ et $Chg_{A_S}(ajouterAuPanier) = \{panier\}$,
- $Chg_{A_S}(dureeLocation) = \{panier\}$ et $Chg_{A_L}(dureeLocation) = \{panier\}$,

Client A_C	Services A_S
$Pre_{\Psi_{A_C}(login)} \equiv id{>}0 \wedge 8{\leq}mdp.length(){\leq}10 \wedge sess = inactive$	$Pre_{\Psi_{A_S}(login)} \equiv id{\geq}1 \wedge 6{\leq}mdp.length(){\leq}10 \wedge sess = inactive$
$Post_{\Psi_{A_C}(login)} \equiv membre.getId() = id$	$Post_{\Psi_{A_S}(login)} \equiv membre.getId() = id$
$Pre_{\Psi_{A_C}(echec)} \equiv errmess.length() \neq 0$	$Pre_{\Psi_{A_S}(echec)} \equiv errmess.length() \neq 0$
$Post_{\Psi_{A_C}(echec)} \equiv vrai$	$Post_{\Psi_{A_S}(echec)} \equiv vrai$
$Pre_{\Psi_{A_C}(connecte)} \equiv sess = inactive$	$Pre_{\Psi_{A_S}(connecte)} \equiv sess = inactive$
$Post_{\Psi_{A_C}(connecte)} \equiv sess = active$	$Post_{\Psi_{A_S}(connecte)} \equiv sess = active$
$Pre_{\Psi_{A_C}(ajouterAuPanier)} \equiv idRes \neq 0 \wedge sess = active$	$Pre_{\Psi_{A_S}(ajouterAuPanier)} \equiv idRes \neq 0 \wedge sess = active$
$Post_{\Psi_{A_C}(ajouterAuPanier)} \equiv panier = nonvide$	$Post_{\Psi_{A_S}(ajouterAuPanier)} \equiv panier = nonvide$
$Pre_{\Psi_{A_C}(validerLoc)} \equiv idMb \neq 0 \wedge idLoc \neq 0 \wedge panier{=}nonvide$	$Pre_{\Psi_{A_S}(validerLoc)} \equiv idMb \neq 0 \wedge idLoc \neq 0 \wedge panier{=}nonvide$
$Post_{\Psi_{A_C}(validerLoc)} \equiv vrai$	$Post_{\Psi_{A_S}(validerLoc)} \equiv vrai$
$Pre_{\Psi_{A_C}(locNonValide)} \equiv errmess.length() \neq 0$	$Pre_{\Psi_{A_S}(locNonValide)} \equiv errmess.length() \neq 0$
$Post_{\Psi_{A_C}(locNonValide)} \equiv vrai$	$Post_{\Psi_{A_S}(locNonValide)} \equiv vrai$
$Pre_{\Psi_{A_C}(locValide)} \equiv vrai$	$Pre_{\Psi_{A_S}(locValide)} \equiv vrai$
$Post_{\Psi_{A_C}(locValide)} \equiv vrai$	$Post_{\Psi_{A_S}(locValide)} \equiv vrai$
$Pre_{\Psi_{A_C}(deconnexion)} \equiv sess = active$	$Pre_{\Psi_{A_S}(dconnexion)} \equiv sess = active$
$Post_{\Psi_{A_C}(deconnexion)} \equiv sess = inactive$	$Post_{\Psi_{A_S}(deconnexion)} \equiv sess = inactive$

Services A_S	Location A_L
$Pre_{\Psi_{A_S}(dureeLocation)} \equiv panier = nonvide \wedge debut.getTime() < fin.getTime()$	$Pre_{\Psi_{A_L}(dureeLocation)} \equiv panier = nonvide \wedge debut.getTime() < fin.getTime()$
$Post_{\Psi_{A_C}(dureeLocation)} \equiv panier = vide$	$Post_{\Psi_{A_S}(dureeLocation)} \equiv panier = vide$

TABLE 6.1 – Sémantique des actions dans *Partagées*(A_C,A_S) et *Partagées*(A_S,A_L).

– le reste des actions de A_C et A_S ne changent aucune variable.

Les prédicats de transition des actions partagées entre A_C et A_S et entre A_S et A_L sont définis dans le tableau 6.2. Le prédicat de transition de l'action *dureeLocation* dans l'automate A_L est défini par $\Omega_{A_L}(dureeLocation) \equiv panier = nonvide \wedge panier' = vide$. ■

Variables totalement et partiellement contrôlées

Nous appelons par les variables *totalement contrôlées* par un automate d'interface sémantique A_c, l'ensemble des variables partagées dans V_{A_c} tel que A_c partage toutes les actions de l'environnement qui changent ces variables. L'ensemble des variables *partiellement contrôlées* par A_c représentent l'ensemble des variables partagées qui peuvent être modifiées par des actions de l'environnement non connues par A_c.

Définition 6.3 (*Variables totalement et partiellement contrôlées*)**.** *L'ensemble* $V^{\text{tc}}_{A_c}$ *des variables* totalement partagées *par* A_c *est défini par* $\{v \in V_{A_c} \mid (\forall c' \in \mathcal{C} \setminus \{c\}, a \in \Sigma_{A_{c'}} \mid v \in Chg_{A_{c'}}(a) \Rightarrow a \in \Sigma^{\text{ext}}_{A_c})\}$. *L'ensemble des variables* partiellement partagées $V^{\text{pc}}_{A_c}$ *est défini par* $V_{A_c} \setminus V^{\text{tc}}_{A_c}$.

Exemple 6.2. La variable *sess* est totalement contrôlée simultanément par A_C et A_S. Par contre, la variable *panier* est totalement contrôlée par A_S et partiellement contrôlée par A_C et A_L. L'action *dureeLocation* qui change la variable *panier* ($panier \in Chg_{A_S}(dureeLocation)$) n'appartient pas à $\Sigma^{\text{ext}}_{A_C}$. Également, la variable *panier* appartient à $Chg_{A_S}(ajouterAuPanier)$ et l'action *ajouterAuPanier* n'appartient pas à $\Sigma^{\text{ext}}_{A_L}$. ■

	Ω_{A_C}	Ω_{A_S}
login	$sess = inactive \wedge Inchangé(LV_{A_C})$	$sess = inactive \wedge ((tents \geq 0 \wedge tents' = tents - 1 \wedge sess' = sess \wedge panier' = panier) \vee Inchangé(LV_{A_S}\}))$
echec	$sess = inactive \wedge Inchangé(LV_{A_C})$	$sess = inactive \wedge Inchangé(LV_{A_S})$
connecte	$sess = inactive \wedge sess' = active \wedge Inchangé(\{panier\})$	$sess = inactive \wedge 0 \leq tents < 2 \wedge sess' = active \wedge Inchangé(\{tents, panier\})$
ajouterAuPanier	$sess = active \wedge ((panier = vide \wedge panier' = nonvide \wedge sess' = sess) \vee Inchangé(LV_{A_C}))$	$sess = active \wedge ((panier = vide \wedge panier' = nonvide \wedge tents' = tents \wedge sess' = sess) \vee Inchangé(LV_{A_S}))$
validerLoc	$sess = active \wedge Inchangé(LV_{A_C})$	$sess = active \wedge Inchangé(LV_{A_S})$
debiter	n'est pas défini	$sess = active \wedge Inchangé(LV_{A_S})$
creditInsuff	n'est pas défini	$sess = active \wedge Inchangé(LV_{A_S})$
nvCredit	n'est pas défini	$sess = active \wedge Inchangé(LV_{A_S})$
dureeLocation	n'est pas défini	$sess = active \wedge panier = nonvide \wedge panier' = vide \wedge tents' = tents \wedge sess' = sess$
locNonValide	n'est pas défini	$sess = active \wedge Inchangé(LV_{A_S})$
locValide	$sess = active \wedge Inchangé(LV_{A_C})$	$sess = active \wedge Inchangé(LV_{A_S})$
deconnexion	$sess = active \wedge sess' = inactive \wedge Inchangé(LV_{A_C} \setminus \{sess\})$	$sess = active \wedge sess' = inactive \wedge tents' = 2$ $Inchangé(LV_{A_S} \setminus \{sess, tents\})$

TABLE 6.2 – Prédicats de transition des actions des SIAs A_C et A_S

6.2 Composabilité, compatibilité et composition

Définition 6.4 (*Composabilité des SIAs*)**.** *Deux automates d'interface sémantiques A_{c_1} et A_{c_2} de deux composants c_1 et c_2 dans $\mathcal{C}$ sont* composables *ssi*

- $\Sigma^{\mathrm{I}}_{A_{c_1}} \cap \Sigma^{\mathrm{I}}_{A_{c_2}} = \Sigma^{\mathrm{O}}_{A_{c_1}} \cap \Sigma^{\mathrm{O}}_{A_{c_2}} = \Sigma^{\mathrm{H}}_{A_{c_1}} \cap \Sigma_{A_{c_2}} = \Sigma_{A_{c_1}} \cap \Sigma^{\mathrm{H}}_{A_{c_2}} = \emptyset$;
- $L_{A_{c_1}} \cap L_{A_{c_1}} = \emptyset$;
- $\mathit{Init}_{A_{c_1}}\langle V_{A_{c_1}} \cap V_{A_{c_2}}\rangle = \mathit{Init}_{A_{c_2}}\langle V_{A_{c_1}} \cap V_{A_{c_2}}\rangle$;
- *Pour toute* $a \in$ *Partagées*(A_{c_1},A_{c_2}) *et* $\phi \in \mathcal{I}_{V_{A_{c_1}} \cap V_{A_{c_2}}}$, $\mathcal{E}(\phi, \Omega_{A_{c_1}}(a)) = \mathcal{E}(\phi, \Omega_{A_{c_2}}(a))$;
- *Pour toute* $a \in$ *Partagées*(A_{c_1},A_{c_2}) *dont la signature est donnée par* $a(i_1,...,i_n) \rightarrow (o)$ *dans* A_{c_1} *et par* $a(i'_1,...,i'_n) \rightarrow (o')$ *dans* A_{c_2} *pour* $n \in \mathbb{N}^*$
  - *si* $a \in \Sigma^{\mathrm{O}}_{A_{c_1}}$, *alors* $D_{i_k} \subseteq D_{i'_k}$ *pour* $1 \leq k \leq n$ *et* $D_o \subseteq D_{o'}$;
  - *si* $a \in \Sigma^{\mathrm{I}}_{A_{c_1}}$, *alors* $D_{i_k} \supseteq D_{i'_k}$ *pour* $1 \leq k \leq n$ *et* $D_o \supseteq D_{o'}$.

La composition de deux automates d'interface sémantiques ne peut prendre effet que si (i) les ensembles de leurs actions d'entrée sont disjoints et de même pour les actions de sortie, et l'ensemble des actions internes de l'un des deux automates est disjoint avec l'ensemble des actions de l'autre, (ii) les ensembles des variables locales de A_{c_1} et A_{c_2} sont disjoints. (iii) pour toute action partagée a, l'effet du prédicat de transition de a dans A_{c_1} ne doit pas contredire celui du prédicat de transition de a dans A_{c_2} sur les variables $V_{A_{c_1}} \cap V_{A_{c_2}}$, (iv) les valuations initiales des variables partagées dans les deux automates sont les mêmes, et (v) la propriété de sous-typage (cf. section 2.3.1) des domaines des paramètres des actions partagées dans *Partagées*(A_{c_1},A_{c_2}) est satisfaite.

La définition de la compatibilité sémantiques entre les automates d'interface sémantiques est

exactement la même que celle présentées dans le chapitre 4 (définition 4.4). Le produit synchronisé de deux automates d'interface sémantique est défini de la manière suivante.

Définition 6.5 (*Produit synchronisé $\otimes$ de deux SIAs*)**.** *Soient A_{c_1} et A_{c_2} deux automates d'interface sémantiques composables, le* produit synchronisé $A_{c_1} \otimes A_{c_2}$ *de A_{c_1} et A_{c_2} est défini par :*

- $S_{A_{c_1}\otimes A_{c_2}} = S_{A_{c_1}} \times S_{A_{c_2}}$ *et* $i_{A_{c_1}\otimes A_{c_2}} = (i_{A_{c_1}}, i_{A_{c_2}})$;
- $\Sigma^{\mathrm{I}}_{A_{c_1}\otimes A_{c_2}} = (\Sigma^{\mathrm{I}}_{A_{c_1}} \cup \Sigma^{\mathrm{I}}_{A_{c_2}}) \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$;
- $\Sigma^{\mathrm{O}}_{A_{c_1}\otimes A_{c_2}} = (\Sigma^{\mathrm{O}}_{A_{c_1}} \cup \Sigma^{\mathrm{O}}_{A_{c_2}}) \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$;
- $\Sigma^{\mathrm{H}}_{A_{c_1}\otimes A_{c_2}} = \Sigma^{\mathrm{H}}_{A_{c_1}} \cup \Sigma^{\mathrm{H}}_{A_{c_2}} \cup \{a \in \mathit{Partagées}(A_{c_1}, A_{c_2}) \mid \mathit{SemComp}_a(A_{c_1}, A_{c_2}) \equiv \mathit{vrai}\}$;
- $L_{A_{c_1}\otimes A_{c_2}} = L_{A_{c_1}} \cup L_{A_{c_2}}$;
- $V_{A_{c_1}\otimes A_{c_2}} = V_{A_{c_1}} \cup V_{A_{c_2}}$;
- $\mathit{Init}_{A_{c_1}\otimes A_{c_2}}$ *égale à* $\mathit{Init}_{A_{c_1}}$ *pour tout* $v \in LV_{A_{c_1}}$ *et* $\mathit{Init}_{A_{c_2}}$ *pour tout* $v \in LV_{A_{c_2}}$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_{c_1}\otimes A_{c_2}}$ *si*
  - $a \notin \mathit{Partagées}(A_{c_1}, A_{c_2}) \wedge (s_1, a, s'_1) \in \delta_{A_{c_1}} \wedge s_2 = s'_2$ *ou*
  - $a \notin \mathit{Partagées}(A_{c_1}, A_{c_2}) \wedge (s_2, a, s'_2) \in \delta_{A_{c_2}} \wedge s_1 = s'_1$ *ou*
  - $a \in \mathit{Partagées}(A_{c_1}, A_{c_2}) \wedge (s_1, a, s'_1) \in \delta_{A_{c_1}} \wedge (s_2, a, s'_2) \in \delta_{A_{c_2}}$ $\wedge\ \mathit{SemComp}_a(A_{c_1}, A_{c_2}) \equiv \mathit{vrai}$;
- $\Psi_{A_{c_1}\otimes A_{c_1}}$ *est définie par :*
  - $\Psi_{A_{c_1}}$ *pour* $a \in \Sigma_{A_{c_1}} \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$;
  - $\Psi_{A_{c_2}}$ *pour* $a \in \Sigma_{A_{c_2}} \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$;
  - $\langle \mathit{Pre}_{\Psi_{A_{c_1}}(a)}, \mathit{Post}_{\Psi_{A_{c_2}}(a)} \rangle$ *pour* $a \in \mathit{Partagées}(A_{c_1}, A_{c_2}) \cap \Sigma^{\mathrm{O}}_{A_{c_1}}$ *telle que* $\mathit{SemComp}_a(A_{c_1}, A_{c_2}) \equiv \mathit{vrai}$;
  - $\langle \mathit{Pre}_{\Psi_{A_{c_2}}(a)}, \mathit{Post}_{\Psi_{A_{c_1}}(a)} \rangle$ *pour* $a \in \mathit{Partagées}(A_{c_1}, A_{c_2}) \cap \Sigma^{\mathrm{I}}_{A_{c_1}}$ *telle que* $\mathit{SemComp}_a(A_{c_1}, A_{c_2}) \equiv \mathit{vrai}$.
- $\mathit{Chg}_{A_{c_1}\otimes A_{c_1}}$ *est définie par :*
  - *pour tout* $a \in \mathit{Partagées}(A_{c_1}, A_{c_2})$, *nous avons* $\mathit{Chg}_{A_{c_1}\otimes A_{c_2}}(a) = \mathit{Chg}_{A_{c_1}}(a) \cup \mathit{Chg}_{A_{c_2}}(a)$;
  - *pour tout* $a \in \Sigma_{A_{c_1}} \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$, $\mathit{Chg}_{A_{c_1}\otimes A_{c_2}}(a) = \mathit{Chg}_{A_{c_1}}(a)$;
  - *pour tout* $a \in \Sigma_{A_{c_2}} \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$, $\mathit{Chg}_{A_{c_1}\otimes A_{c_2}}(a) = \mathit{Chg}_{A_{c_1}}(a)$;
- $\Omega_{A_{c_1}\otimes A_{c_1}}$ *est définie par :*
  - *pour tout* $a \in \mathit{Partagées}(A_{c_1}, A_{c_2})$, $\Omega_{A_{c_1}\otimes A_{c_2}}(a) = \Omega_{A_{c_1}}(a) \wedge \Omega_{A_{c_2}}(a)$;
  - *pour tout* $a \in \Sigma_{A_{c_1}} \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$, $\Omega_{A_{c_1}\otimes A_{c_2}}(a) = \Omega_{A_{c_1}}(a)$;
  - *pour tout* $a \in \Sigma_{A_{c_2}} \setminus \mathit{Partagées}(A_{c_1}, A_{c_2})$, $\Omega_{A_{c_1}\otimes A_{c_2}}(a) = \Omega_{A_{c_2}}(a)$.

Théorème 6.1. *La produit synchronisé $\otimes$ des automates d'interface sémantiques composables est une opération associative.*

Démonstration. La preuve est exactement la même que celle du théorème 4.1 sauf qu'il faut prouver en plus, que, pour tous automates d'interface sémantiques A_1, A_2 et A_3 mutuellement composables, $L_{A_1\otimes A_2\otimes A_3}$, $V_{A_1\otimes A_2\otimes A_3}$, $\mathit{Chg}_{A_1\otimes A_2\otimes A_3}$, $\Omega_{A_1\otimes A_2\otimes A_3}$, et $\mathit{Init}_{A_1\otimes A_2\otimes A_3}$ sont toujours les mêmes pour n'importe quelle application des parenthèses. Le lecteur peut déduire ça aisément en se basant sur la définition 6.5. □

Les définitions des états illégaux, des états compatibles et la compatibilité entre deux automates d'interface sémantiques sont exactement les mêmes que celles données dans le chapitre 4.

Définition 6.6 (*Composition* $\|$ *des SIAs*)**.** *La* composition $A_{c_1} \parallel A_{c_2}$ *de deux automates d'interface sémantiques, est définie par*

- $S_{A_{c_1}\|A_{c_2}} = Comp(A_{c_1}, A_{c_2})$;
- $i_{A_{c_1}\|A_{c_2}} = \{i_{A_{c_1}\otimes A_{c_2}}\} \cap Comp(A_{c_1}, A_{c_2})$;
- $\Sigma^{\mathrm{I}}_{A_{c_1}\|A_{c_2}} = \Sigma^{\mathrm{I}}_{A_{c_1}\otimes A_{c_2}}$, $\Sigma^{\mathrm{O}}_{A_{c_1}\|A_{c_2}} = \Sigma^{\mathrm{O}}_{A_{c_1}\otimes A_{c_2}}$, *et* $\Sigma^{\mathrm{H}}_{A_{c_1}\|A_{c_2}} = \Sigma^{\mathrm{H}}_{A_{c_1}\otimes A_{c_2}}$;
- $\delta_{A_{c_1}\|A_{c_2}} = \delta_{A_{c_1}\otimes A_{c_2}} \cap (Comp(A_{c_1}, A_{c_2}) \times \Sigma_{A_{c_1}\|A_{c_2}} \times Comp(A_{c_1}, A_{c_2}))$;
- $L_{A_{c_1}\|A_{c_2}} = L_{A_{c_1}\otimes A_{c_2}}$;
- $V_{A_{c_1}\|A_{c_2}} = V_{A_{c_1}\otimes A_{c_2}}$;
- $\mathsf{Init}_{A_{c_1}\|A_{c_2}} = \mathsf{Init}_{A_{c_1}\otimes A_{c_2}}$;
- $\Psi_{A_{c_1}\|A_{c_2}} = \Psi_{A_{c_1}\otimes A_{c_2}}$;
- $Chg_{A_{c_1}\|A_{c_2}} = Chg_{A_{c_1}\otimes A_{c_2}}$;
- $\Omega_{A_{c_1}\|A_{c_2}} = \Omega_{A_{c_1}\otimes A_{c_2}}$.

Théorème 6.2. *La composition* $\|$ *des automates d'interface sémantiques composables est une opération associative.*

Démonstration. La preuve est triviale en se basant sur celles des théorèmes 6.1 et 4.2. □

Exemple 6.3. Selon le tableau 6.1, A_C, A_S et A_L sont mutuellement compatibles et la composition finale $A_C \parallel A_S \parallel A_L$ entre les trois est non vide. Le lecteur peut se référer aux figures 4.3 et 4.4 pour déduire l'automate d'interface sémantique $A_C \parallel A_S \parallel A_L$.

Si $SemComp_{validerLoc}(A_C, A_S) \equiv \mathsf{faux}$ ou $SemComp_{login}(A_C, A_S) \equiv \mathsf{faux}$ par exemple, A_C et A_S deviennent incompatibles et par conséquent $A_C \parallel A_S \parallel A_L$ est vide. ■

6.3 Raffinement

Dans cette section, nous adaptons la relation de raffinement présentée dans le chapitre 4 pour les automates d'interface sémantiques. Les définitions de la substitution sémantique et la simulation alternée des automates d'interface sémantiques restent les mêmes que celles présentées dans le chapitre 4. Nous allons uniquement reformuler la définition 4.9 pour supporter l'utilisation des variables.

Le raffinement d'un composant peut être utilisé dans un environnement plus grand, des nouveaux composants peuvent être ajoutés à l'ensemble $\mathcal{C}$ des composants. Dans ce contexte, supposons qu'on veut raffiner un composant $c \in \mathcal{C}$, nous devons définir un nouvel ensemble de composants $\mathcal{C}' \supseteq \mathcal{C}$. Par conséquent, l'ensemble de variables $V_{\mathcal{C}}$ est inclus dans $V_{\mathcal{C}'}$. En se basant sur les définitions 4.8 et 3.10 et les préliminaires introductifs présentés dans la section 3.5, la définition du raffinement est adaptée est présentée ci-dessous :

Définition 6.7 (*Raffinement des SIAs*)**.** *Le* raffinement A'_c *d'un automate d'interface sémantique* A_c*, noté* $A'_c \preceq A_c$*, satisfait les conditions suivantes :*

1. $\Sigma^{\mathrm{I}}_{A_c} \subseteq \Sigma^{\mathrm{I}}_{A'_c}$ *et* $\Sigma^{\mathrm{O}}_{A_c} \supseteq \Sigma^{\mathrm{O}}_{A'_c}$;
2. *Pour toute* $a \in \Sigma^{\mathrm{ext}}_{A_c} \cap \Sigma^{\mathrm{ext}}_{A'_c}$, $SemSub_a(A_c, A'_c) \equiv \mathsf{vrai}$;

3. $L_{A'_c} \supseteq L_{A_c}$;
4. $V_{A'_c} \supseteq V_{A_c}$, $V^{\mathrm{tc}}_{A'_c} \supseteq V^{\mathrm{tc}}_{A_c}$ *et* $V^{\mathrm{pc}}_{A'_c} \supseteq V^{\mathrm{pc}}_{A_c}$;
5. $\mathit{Init}_{A'_c}\langle LV_{A_c}\rangle = \mathit{Init}_{A_c}$;
6. *Pour toute* $a \in \Sigma_{A_c} \cap \Sigma_{A'_c}$ *et* $\phi \in \mathcal{I}_{LV_{A_c}}$, $\mathcal{E}(\phi, \Omega_{A'_c}(a))$ *égale à* $\mathcal{E}(\phi, \Omega_{A_c}(a))$ *ou* ϕ ;
7. *Pour toute* $a \in \Sigma_{A'_c} \setminus \Sigma_{A_c}$ *et* $\phi \in \mathcal{I}_{LV_{A_c}}$, $\mathcal{E}(\phi, \Omega_{A'_c}(a))$ *égal à* ϕ ;
8. *Il existe une simulation alternée* $\preceq$ *entre* A'_c *et* A_c *tel que* $i_{A'_c} \preceq i_{A_c}$ *où* $i \in I_{A_c}$ *et* $i' \in I_{A'_c}$ *sont respectivement les états initiaux de* A_c *et* A'_c.

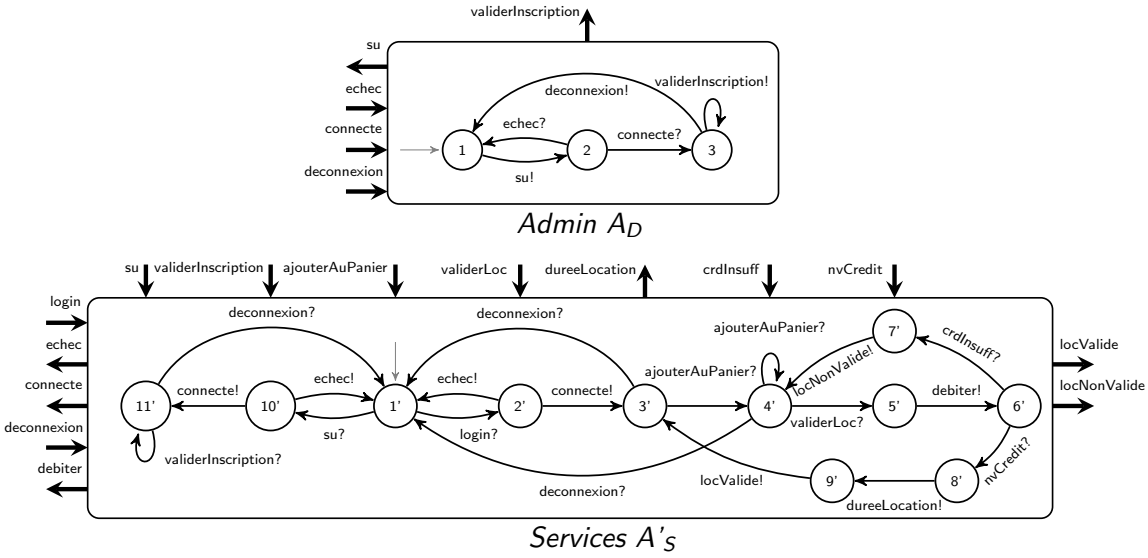


FIGURE 6.2 – Automates d'interface sémantique de A_D du composant Admin et le raffinement A'_S de A_S

Selon la définition, l'ensemble des variables partagées du raffinement A'_c doit inclure l'ensemble des variables de A_c. Une variable totalement contrôlée (respectivement partiellement partagée) par A_c l'est aussi dans A'_c. Les prédicats de transition d'une action $a \in \Sigma_{A_c} \cap \Sigma_{A'_c}$ dans A'_c et A_c doivent avoir les mêmes effets sur les variables LV_{A_c} si a change ces variables. Sinon, ces variables restent inchangées après l'exécution de a dans A_c.

Exemple 6.4. Nous reprenons l'exemple 4.7 présenté dans le chapitre 4 et l'exemple 6.1. L'ensemble $\mathcal{C}'$ est défini par {Client,Services,Admin,LocationRessource}. L'ensemble des variables $V_{\mathcal{C}'} = V_{\mathcal{C}} \cup \{asess\}$. La variable *asess* dont le domaine est $D_{asess} = \{active, inactive\}$, indique le statut de la session administrateur. L'ensemble des variables $V_{A'_S}$ de A'_S, montré dans la figure 6.2, égale à $V_{A_S} \cup \{asess\}$.

L'ensemble des variables $L_{A'_S}$ égal à L_{A_S}. L'ensemble des variables LV_{A_D} de A_D (cf. figure 6.2) égale à $\{asess\}$. Les actions qui modifient la nouvelle variable *asess* sont *connecte* et *deconnexion*. Nous assumons que $\mathit{Init}_{A_S}\langle LV_{A'_S}\rangle = \{sess \mapsto inactive, tents \mapsto 2, panier \mapsto vide, asess \mapsto inactive\}$. Nous présentons uniquement dans cet exemple les prédicats de transition des actions *deconnexion* et *connecte* dans A'_S. Le prédicat de transition $\Omega_{A'_S}(connecte)$ de l'action *connecte* dans A'_S est défini par

$$sess = inactive \wedge asess = inactive \wedge$$
$$((0 \leq tents < 2 \wedge sess' = active \wedge asess' = asess) \vee (asess' = active \wedge sess' = sess))$$
$$\wedge tents' = tents \wedge panier' = panier$$

Le prédicat $\Omega_{A_D}(connecte)$ est défini par $asess = inactive \wedge asess' = active$. Le prédicat $\Omega_{A_D}(deconnexion)$ est défini par $asess = active \wedge asess' = inactive$ et dans A'_S. Le prédicat $\Omega_{A'_S}(deconnexion)$ est défini par

$$((sess = inactive \wedge asess = active \ \wedge \ asess' = inactive \wedge sess' = sess) \vee$$
$$(sess = active \wedge asess = inactive \wedge tents' = 2 \wedge sess' = inactive \wedge asess' = asess))$$
$$\wedge panier' = panier.$$

Nous pouvons remarquer que $\mathsf{Init}_{A'_S}\langle LV_{A_S}\rangle = \mathsf{Init}_{A_S}$. La condition 6 de la définition 6.7 est satisfaite pour les deux actions *deconnexion* et *connecte*. L'automate A'_S raffine A_S ($A'_S \preceq A_S$) si les conditions 2, 6 et 7 de la définition 6.7 sont satisfaites. ■

Théorème 6.3. *Soient trois automates d'interface sémantiques A, A' et E tels que A' et E sont composables et $\Sigma^{\mathrm{I}}_{A'} \cap \Sigma^{\mathrm{O}}_{E} \subseteq \Sigma^{\mathrm{I}}_{A} \cap \Sigma^{\mathrm{O}}_{E}$. Si $A \sim E$ et $A' \preceq A$, alors $A' \sim E$ et $A' \,\|E \preceq A \,\|E$.*

Démonstration. La preuve est triviale en se basant sur les définitions 6.7 et 6.4 et la preuve du théorème 4.3. □

6.4 Vérification de propriétés d'invariance

Dans cette section, nous définissons les aspects formels utilisés pour spécifier et vérifier les propriétés de sûreté des automates d'interface sémantiques en se basant sur la sémantique des actions et l'utilisation des variables. Les propriétés de sûreté sont typiquement des *invariants* définis sur les variables partagées. Communément, pour vérifier les propriétés d'un système, il faut le spécifier avec un système de transitions. Les états de ce système de transitions sont associés à des propositions atomiques définies sur des variables. Notre approche est basée sur une procédure similaire. Un automate d'interface sémantique A est transformé à un système de transitions étiquetées $\mathsf{LTS}(A)$ dont les états sont les valuations des variables. En partant de la valuation initiale des variables et en suivant l'ordonnancement des transitions dans δ_A, les prédicats de transition des actions modifient les états et le système de transitions $\mathsf{LTS}(A)$ peut être généré.

6.4.1 Représentation LTS d'un automate d'interface sémantique

La représentation LTS d'un automate d'interface sémantique A transforme son ensemble des transitions à un système de transitions étiquetées dont les états sont définis par $\mathcal{I}_{LV_A}$ et les étiquettes sont les actions Σ_A. Les représentations LTS des automates d'interface sémantiques permettent de séparer la tâche de vérification d'interopérabilité des automates d'interface sémantiques de la tache de vérification de propriétés. Il est clair qu'un automate d'interface sémantique contient moins d'états que sa représentation LTS ce qui permet de réduire la complexité de vérification de compatibilité et du calcul du raffinement. Les représentations LTS sont consacrées pour vérifier les propriétés des automates d'interface sémantiques (cf. section 2.4).

En se basant sur la définition 6.2, nous définissons la représentation LTS d'un automate d'interface sémantique de la manière suivante.

Définition 6.8 (*Représentation LTS*). *Une* représentation LTS $\mathit{LTS}(A) = \langle S_{\mathit{LTS}(A)}, I_{\mathit{LTS}(A)}, \Sigma_{\mathit{LTS}(A)}, \delta_{\mathit{LTS}(A)} \rangle$ *d'un automate d'interface sémantique* A *est un système de transitions étiquetées défini par*

- $S_{\mathit{LTS}(A)} \subseteq \mathcal{I}_{LV_A}$;
- $I_{\mathit{LTS}(A)} = \mathit{Init}_A$;
- $\Sigma_{\mathit{LTS}(A)} = \Sigma_A$;
- $(\phi_1, a, \phi_2) \in \delta_{\mathit{LTS}(A)}$ *ssi* $\phi_1 \in \mathcal{I}^s_{LV_A}$, *pour* $s \in S_A$ *et* $a \in \Sigma_A(s)$, *et* $\phi_2 = \mathcal{E}(\phi_1, \Omega_A(a))$.

Exemple 6.5. La représentation LTS $\mathit{LTS}(A_C)$ de l'automate A_C du composant Client est montrée dans la figure 6.3.

Les exécutions valides de la représentation LTS $\mathit{LTS}(A)$ d'un SIA A sont celles qui respectent les contraintes du protocole spécifiées par A. Ces contraintes doivent être prises en compte lors de la vérification des propriétés temporelles sur le LTS. Par exemple, l'exécution qui active plusieurs fois la séquence *echec*,*login* n'est pas valide car le protocole exige que l'action *login* doit être activée avant *echec*. ■

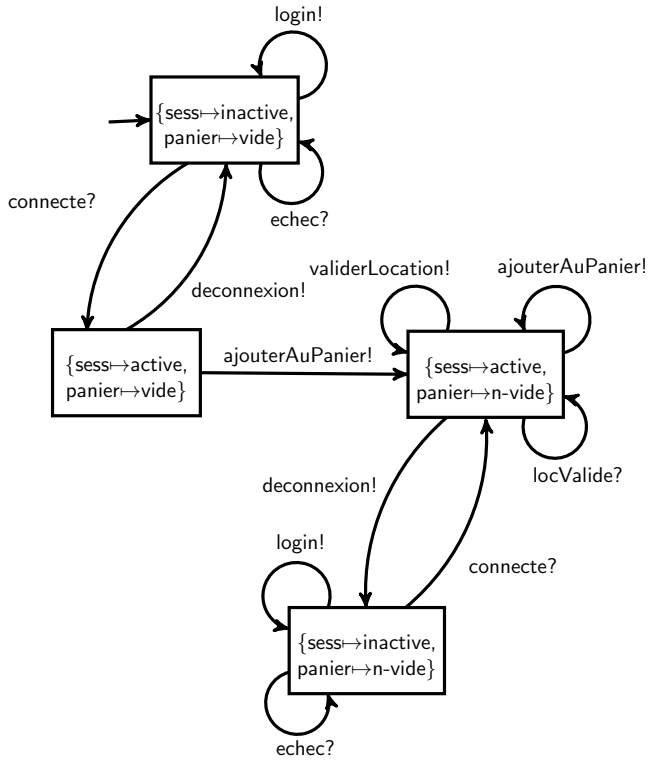


FIGURE 6.3 – La représentation LTS $\mathit{LTS}(A_C)$ de A_C

Nous notons par LV^{tc}_A, l'ensemble $V^{\mathrm{tc}}_A \cup L_A$ d'un automate d'interface sémantique A.

Propriété 6.1. *Soient* $W = LV_{A_1 \| A_2}$, $V = LV^{\mathrm{tc}}_{A_1} \cap LV^{\mathrm{tc}}_{A_2}$, $V' = LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}$, $V_1 = LV^{\mathrm{tc}}_{A_1} \setminus LV^{\mathrm{tc}}_{A_2}$, *et* $V_2 = LV^{\mathrm{tc}}_{A_2} \setminus LV^{\mathrm{tc}}_{A_1}$ *où* A_1 *et* A_2 *sont deux automates d'interface sémantiques compatibles, pour tout* $\phi \in S_{\mathit{LTS}(A_1 \| A_2)}$, *il existe* $\phi_1 \in S_{\mathit{LTS}(A_1)}$ *et* $\phi_2 \in S_{\mathit{LTS}(A_2)}$ *telles que*

1. $\phi\langle V \rangle = \phi_1\langle V \rangle = \phi_2\langle V \rangle$ *sachant que* $(V = LV^{\mathrm{tc}}_{A_1} \cap LV^{\mathrm{tc}}_{A_2} = V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2})$;

2. $\phi\langle V'\rangle$ *égale à la fonction suivante :*

$$\begin{cases} \phi\langle V\rangle & \textit{pour tout } v \in V; \\ \phi_1\langle V_1\rangle & \textit{pour tout } v \in V_1; \\ \phi_2\langle V_2\rangle & \textit{pour tout } v \in V_2. \end{cases}$$

Cette propriété peut être déduite facilement à partir des définitions 6.3, 6.4, 6.5 et 6.8. La propriété montre que, pour chaque état $\phi \in S_{\mathsf{LTS}(A_1 \| A_2)}$, uniquement les valuations des variables totalement contrôlées simultanément par A_1 et A_2 sont les mêmes dans A_1, A_2 et $A_1 \parallel A_2$. Ces variables sont modifiables uniquement par des actions dans $\mathit{Partagées}(A_1, A_2)$.

6.4.2 Spécification des invariants

Soit une représentation LTS $\mathsf{LTS}(A)$ d'un automate d'interface sémantique A, nous dénotons par $\varphi[V] \in \mathit{Preds}(V)$ (cf. section 4.1) une formule de premier ordre où toutes les variables libres utilisées dans $\varphi[V]$ sont des variables dans $V \subseteq LV_A$. Une formule $\varphi[V]$ est satisfaite à l'état $\phi \in S_{\mathsf{LTS}(A)}$ ssi l'état ϕ satisfait $\varphi[V]$, i.e. la condition suivante est satisfaite

$$(\bigwedge_{v \in LV_A} v = \phi(v)) \Rightarrow \varphi[V].$$

Définition 6.9 (*Invariant*)**.** *Soit une représentation LTS* $\mathsf{LTS}(A)$ *d'un automate d'interface sémantique* A *et un ensemble* $V \subseteq LV_A$*, un* invariant $\varphi[V]$ *de* $\mathsf{LTS}(A)$ *(écrit* $\mathsf{LTS}(A) \models \varphi[V]$*) est une formule logique de premier ordre tel que pour tout* $\phi \in S_{\mathsf{LTS}(A)}$*,* $\phi\langle V\rangle$ *satisfait* $\varphi[V]$ *(*$\phi\langle V\rangle \models \varphi[V]$*) et ainsi* ϕ *satisfait* $\varphi[V]$ *(*$\phi \models \varphi[V]$*).*

Exemple 6.6. Le prédicat $\varphi[LV_{A_S}] = \neg(\mathit{tents} = 2 \wedge \mathit{sess} = \mathit{active})$ est un invariant de $\mathsf{LTS}(A_S)$ montré dans la figure 6.3. Le prédicat $\varphi[LV_{A'_S}] = \neg(\mathit{sess} = \mathit{active} \wedge \mathit{asess} = \mathit{active})$ est un invariant de $\mathsf{LTS}(A'_S)$ de A'_S montré dans la figure 6.2. Cet invariant montre qu'une instance du composant Services ne peut pas être connectée à la fois avec une instance du composant Client et une instance du composant Admin. ■

6.4.3 Préservation des invariants par la composition

Le théorème suivant établit que les invariants décrits en fonction des variables totalement contrôlées et locales $LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_1}$ de deux automates d'interface sémantiques A_1 et A_2 composables et compatibles, sont les seuls qui sont préservés par la représentation LTS de la composition $A_1 \parallel A_2$. L'un des deux automates partage toutes les actions de l'autre qui changent ses variables totalement contrôlées. De plus, une action partagée entre les deux automates agit de la même façon sur ces variables ce qui assure la préservation des invariants par composition (cf. définition 6.4). Pareillement, les invariants définis en fonction des variables locales des deux automates sont préservés par la représentation LTS de la composition $A_1 \parallel A_2$ car $L_{A_1} \cap L_{A_1} = \emptyset$. Un invariant décrit en fonction des variables partiellement partagées de A_1 ou de A_2 ne peut pas être préservé par la représentation LTS de leur composition. Ceci est dû à la possibilité que A_1 (respectivement A_2) modifie les variables partiellement partagées de A_2 (respectivement A_1) par des actions qui n'appartiennent pas Σ_{A_2} (respectivement Σ_{A_1}) de telle sorte que les invariants établis sur ces variables soient violés.

Théorème 6.4 (*Préservation d'invariant par composition*)**.** *Soient deux représentations* $\mathsf{LTS}(A_1)$ *et* $\mathsf{LTS}(A_2)$ *respectivement de deux automates d'interface sémantiques compatibles* A_1 *et* A_2*, pour tout* $\varphi_1[LV^{\mathrm{tc}}_{A_1}]$ *et* $\varphi_2[LV^{\mathrm{tc}}_{A_2}]$*, si* $\mathsf{LTS}(A_1) \models \varphi_1[LV^{\mathrm{tc}}_{A_1}]$ *et* $\mathsf{LTS}(A_2) \models \varphi_2[LV^{\mathrm{tc}}_{A_2}]$*, alors* $\mathsf{LTS}(A_1 \parallel A_2) \models (\varphi_1[LV^{\mathrm{tc}}_{A_1}] \wedge \varphi_2[LV^{\mathrm{tc}}_{A_2}])$.

Démonstration. Nous avons les hypothèses suivantes :

1. $\forall \phi_1 \in S_{\mathsf{LTS}(A_1)} \mid \phi_1\langle LV^{\mathrm{tc}}_{A_1}\rangle \models \varphi_1[LV^{\mathrm{tc}}_{A_1}]$ et $\phi_1 \models \varphi_1[LV^{\mathrm{tc}}_{A_1}]$;
2. $\forall \phi_2 \in S_{\mathsf{LTS}(A_2)} \mid \phi_2\langle LV^{\mathrm{tc}}_{A_2}\rangle \models \varphi_2[LV^{\mathrm{tc}}_{A_2}]$ et $\phi_2 \models \varphi_2[LV^{\mathrm{tc}}_{A_2}]$;

Nous devons prouver que, pour tout $\phi \in S_{\mathsf{LTS}(A_1 \| A_2)}$, $\phi\langle LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}\rangle \models (\varphi_1[LV^{\mathrm{tc}}_{A_1}] \wedge \varphi_2[LV^{\mathrm{tc}}_{A_2}])$ et $\phi \models (\varphi_1[LV^{\mathrm{tc}}_{A_1}] \wedge \varphi_2[LV^{\mathrm{tc}}_{A_2}])$? Selon la propriété 6.1(2), nous avons pour tout $\phi \in S_{\mathsf{LTS}(A_1 \| A_2)}$, il existe $\phi_1 \in S_{\mathsf{LTS}(A_1)}$ et $\phi_2 \in S_{\mathsf{LTS}(A_2)}$ tels que :

$$\phi\langle LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}\rangle = \begin{cases} \phi\langle LV^{\mathrm{tc}}_{A_1} \cap LV^{\mathrm{tc}}_{A_2}\rangle & \textit{pour tout } v \in LV^{\mathrm{tc}}_{A_1} \cap LV^{\mathrm{tc}}_{A_2}; \\ \phi_1\langle LV^{\mathrm{tc}}_{A_1} \setminus LV^{\mathrm{tc}}_{A_2}\rangle & \textit{pour tout } v \in LV^{\mathrm{tc}}_{A_1} \setminus LV^{\mathrm{tc}}_{A_2}; \\ \phi_2\langle LV^{\mathrm{tc}}_{A_2} \setminus LV^{\mathrm{tc}}_{A_1}\rangle & \textit{pour tout } v \in LV^{\mathrm{tc}}_{A_2} \setminus LV^{\mathrm{tc}}_{A_1}. \end{cases}$$

Nous pouvons déduire, selon la propriété 6.1(1), que

$$\phi\langle LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}\rangle = \begin{cases} \phi_1\langle LV^{\mathrm{tc}}_{A_1}\rangle & \textit{pour tout } v \in LV^{\mathrm{tc}}_{A_1}; \\ \phi_2\langle LV^{\mathrm{tc}}_{A_2}\rangle & \textit{pour tout } v \in LV^{\mathrm{tc}}_{A_2}. \end{cases}$$

Nous avons $\phi_1\langle LV^{\mathrm{tc}}_{A_1}\rangle \models \varphi_1[LV^{\mathrm{tc}}_{A_1}]$ (hypothèse 1) et $\phi_2\langle LV^{\mathrm{tc}}_{A_2}\rangle \models \varphi_2[LV^{\mathrm{tc}}_{A_2}]$ (hypothèse 2), alors nous pouvons déduire que $\phi\langle LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}\rangle \models \varphi_1[LV^{\mathrm{tc}}_{A_1}]$ et $\phi\langle LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}\rangle \models \varphi_2[LV^{\mathrm{tc}}_{A_2}]$. Par conséquent, $\phi\langle LV^{\mathrm{tc}}_{A_1} \cup LV^{\mathrm{tc}}_{A_2}\rangle \models (\varphi_1[LV^{\mathrm{tc}}_{A_1}] \wedge \varphi_2[LV^{\mathrm{tc}}_{A_2}])$ et $\phi \models (\varphi_1[LV^{\mathrm{tc}}_{A_1}] \wedge \varphi_2[LV^{\mathrm{tc}}_{A_2}])$. □

Le corollaire suivant peut être déduit du théorème précédent.

Corollaire 6.1. *Soient deux représentations* $\mathsf{LTS}(A_1)$ *et* $\mathsf{LTS}(A_2)$ *respectivement de deux automates d'interface sémantiques compatibles* A_1 *et* A_2*, pour tout* $\varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$*, si* $\mathsf{LTS}(A_1) \models \varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$ *et* $\mathsf{LTS}(A_2) \models \varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$*, alors* $\mathsf{LTS}(A_1 \parallel A_2) \models \varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$.

Démonstration. Nous avons les hypothèses $A_1 \sim A_2$, $\mathsf{LTS}(A_1) \models \varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$ et $\mathsf{LTS}(A_2) \models \varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$. Selon le théorème précédent, nous pouvons trivialement déduire que $\mathsf{LTS}(A_1 \parallel A_2) \models \varphi[V^{\mathrm{tc}}_{A_1} \cap V^{\mathrm{tc}}_{A_2}]$. □

Exemple 6.7. Le prédicat $\varphi[LV^{\mathrm{tc}}_{A_S}] = \neg(\mathit{tents} = 2 \wedge \mathit{sess} = \mathit{active})$ est un invariant de $\mathsf{LTS}(A_S)$. Nous pouvons déduire que $\mathsf{LTS}(A_C \parallel A_S) \models \neg(\mathit{tents} = 2 \wedge \mathit{sess} = \mathit{active})$. ■

6.4.4 Préservation des invariants par le raffinement

Le théorème suivant établit que tous les invariants d'une représentation LTS d'un automate d'interface sémantique A décrits en fonction des variables partagées de A sont préservés par les représentations LTS de tout raffinement A' de A s'il existe une relation de simulation entre $\mathsf{LTS}(A)$ et $\mathsf{LTS}(A')$: toute valuation des variables appartenant à l'ensemble LV_A à chaque état de $\mathsf{LTS}(A')$ doit correspondre à une valuation du même ensemble LV_A au moins à un état de $\mathsf{LTS}(A)$.

Définition 6.10 (*Relation de simulations entre LTSs*)**.** *Soient les représentations LTS* $\mathsf{LTS}(A)$ *et* $\mathsf{LTS}(A')$ *respectivement de deux automates d'interface sémantiques* A *et* A' *tels que* $A' \preceq A$*, nous disons que* $\mathsf{LTS}(A)$ simule $\mathsf{LTS}(A')$*, noté* $\mathsf{LTS}(A) \rhd \mathsf{LTS}(A')$*, ssi pour tout* $\phi' \in S_{\mathsf{LTS}(A')}$*, il existe* $\phi \in S_{\mathsf{LTS}(A)}$ *tel que* $\phi'\langle LV_A\rangle = \phi$.

Théorème 6.5 (*Préservation d'invariant par raffinement*)**.** *Soient les représentations LTS* $\mathsf{LTS}(A)$ *et* $\mathsf{LTS}(A')$ *respectivement de deux automates d'interface sémantiques* A *et* A' *tels que* $A' \preceq A$ *et* $\mathsf{LTS}(A) \rhd \mathsf{LTS}(A')$*, pour tout* $\varphi[LV_A]$ *et* $\varphi'[LV_{A'} \setminus LV_A]$*, si* $\mathsf{LTS}(A) \models \varphi[LV_A]$ *et* $\mathsf{LTS}(A') \models \varphi'[LV_{A'} \setminus LV_A]$*, alors* $\mathsf{LTS}(A') \models (\varphi[LV_A] \wedge \varphi'[V_{A'} \setminus V_A])$.

Démonstration. Nous avons les hypothèses suivantes :

1. $\forall \phi \in S_{\mathsf{LTS}(A)} \mid \phi \models \varphi[LV_A]$;
2. $\forall \phi' \in S_{\mathsf{LTS}(A')} \mid \phi'\langle V_{A'} \setminus V_A\rangle \models \varphi'[LV_{A'} \setminus LV_A]$;
3. $\forall \phi' \in S_{\mathsf{LTS}(A')} \mid (\exists \phi \in S_{\mathsf{LTS}(A)} \mid \phi'\langle LV_A\rangle = \phi)$ (car $\mathsf{LTS}(A) \rhd \mathsf{LTS}(A')$).

Nous devons prouver que $\forall \phi' \in S_{\mathsf{LTS}(A')} \mid \phi' \models (\varphi[LV_A] \wedge \varphi'[LV_{A'} \setminus LV_A])$? Nous avons $\phi'\langle LV_{A'} \setminus LV_A\rangle \models \varphi'[LV_{A'} \setminus LV_A]$ (hypothèse 2). Nous pouvons déduire que (i) $\phi' \models \varphi'[LV_{A'} \setminus LV_A]$ parce que $(LV_{A'} \setminus LV_A) \subseteq LV_{A'}$. Il reste à prouver que $\phi' \models \varphi[LV_A]$?

Prouver que $\phi' \models \varphi[LV_A]$ est équivalent à prouver que $\phi'\langle LV_A\rangle \models \varphi[LV_A]$. Nous pouvons constater qu'il existe $\phi \in S_{\mathsf{LTS}(A)}$ tel que $\phi'\langle LV_A\rangle = \phi$ (hypothèse 3). Selon l'hypothèse 1, nous pouvons déduire que $\phi'\langle LV_A\rangle \models \varphi[LV_A]$ et par conséquent (ii) $\phi'\langle LV_{A'}\rangle \models \varphi[LV_A]$. Les deux résultat (i) et (ii) impliquent que $\phi' \models (\varphi'[LV_{A'} \setminus LV_A] \wedge \varphi[LV_A])$. □

Le corollaire suivant peut être déduit du théorème précédent.

Corollaire 6.2. *Pour tout prédicat $\varphi[V]$ tel que $V \subseteq (LV_A)$ et $\mathsf{LTS}(A) \models \varphi[V]$, $\mathsf{LTS}(A') \models \varphi[V]$ si les hypothèses du théorème 6.5 sont satisfaites.*

Démonstration. Nous avons $\mathsf{LTS}(A) \models \varphi[V]$ et $\mathsf{LTS}(A') \models \mathit{vrai}$. Selon le théorème précédent, nous pouvons trivialement déduire que $\mathsf{LTS}(A') \models \varphi[V]$. □

Exemple 6.8. La formule $\varphi[LV_{A_S}] = \neg(\mathit{tents} = 2 \wedge \mathit{sess} = \mathit{active})$ est simultanément un invariant de la représentation LTS de l'automate A_S et celle de l'automate A'_S présenté dans la figure 6.2. ■

6.5 Automates d'interface sémantiques et l'implémentation des composants

Dans cette section, nous allons montrer comment utiliser les automates d'interface sémantiques pour générer une partie de l'implémentation des composants. Nous allons prendre comme exemple les composants EJBs de notre étude de cas présentée tout au long du chapitre.

Dans le contexte des EJBs et la programmation orientée objet en général, la notion de variables partagées et globales entre les composants et les objets est souvent n'est pas utilisée. Les variables déclarées au niveau des spécifications automates d'interface sémantiques des composants doivent être traduites autrement au niveau de l'implémentation. Une variable partagée entre deux automates d'interface sémantiques de deux composants est représentée par deux attributs privés dans l'implémentation de chaque composant. Nous allons prendre comme exemple les composants Client et Services et l'automate d'interface sémantique A_C présenté dans la figure 6.1 et A'_S présenté dans la figure 6.2.

Le code suivant représente l'implémentation du Client. Les pré et post-conditions des méthodes et l'invariant sont décrites en JML.

```
public enum SessionStatut { ACTIVE, INACTIVE } ;
public enum PanierStatut { VIDE, NONVIDE } ;

public class Client {
   private SessionStatut sess = INACTIVE ;
   private Panier Statut panier = VIDE ;
   private Membre connecte ;
```

```
   @EJB()
   private ServiceVisiteur sv ;
   @EJB()
   private ServiceMembre sm ;


   /*@ requires id > 0 && mdp.length() >= 8 && mdp.length() <= 10
       && sess == INACTIVE ; @*/
   //@ ensures connecte.getId() = id && sess == ACTIVE
   public void login(int id, String mdp) throws  EchecConnexionExp {
        try{
          connecte = sv.login(id, mdp) ;
          sess = ACTIVE ;
        }
        catch(EchecConnexionExp e) {
            throw new EchecConnexionExp(e);
        }
   }

   //@ requires sess == ACTIVE ;
   //@ ensures sess == INACTIVE ;
   public void deconnexion() {
      connecte = sm.deconnexion() ;
      sess = INACTIVE ;
   }

   //@ requires idRes != 0 && sess == ACTIVE ;
   //@ ensures panier == NONVIDE ;
   public void ajouterAuPanier(int idRes) {
      if(panier == VIDE)
         panier = NONVIDE ;
      sm.ajouterAuPanier(idRes) ;
   }

   //@ requires idMb != 0 && idLoc != 0 && sess == ACTIVE
   //@ && panier == NONVIDE ;
   //@ ensures true;
   public void validerLocation(int idMb, int idLoc) {
      // L'exception LocNonValide n'est pas traitée
      boolean locValide = sm.validerLoc(idMb,idLoc) ;
   }
}
```

L'implémentation du composant Client montre comment peut-on utiliser les automates d'interface sémantiques pour générer du code Java. Nous remarquons que chaque variable dans V_{A_C} est traduite à un attribut privé du composant. Les prédicats de transition des actions sont traduits par les mises à jour des attributs par les méthodes. Nous supposons que le composant Client est invoqué par un programme *main* client ou des *servlets* Java. Les pré et post-conditions de

la méthode *login* représente les pré et post-conditions de l'appel de la méthode *login* `connecte = sv.login(id, mdp)` dans le composant Services qui concrétise les actions *login* ! et *connecte* ? dans A_C. Nous pouvons remarquer de même que l'exception de la méthode `validerLoc` du composant Service n'est pas traitée (absence de `try`/`catch`).

Nous présentons maintenant le code du composant Service en tenant en compte la spécification du automate d'interface sémantique A'_S présentée dans la figure 6.2.

```
@Stateful
public class Services implements ServiceVisiteur, ServiceMembre {
   private SessionStatut sess = INACTIVE ;
   private SessionStatut asess = INACTIVE ;
   private int tents = 2 ;
   private PanierStatut panier = VIDE ;

   @PersistenceContext ...
   private EntityManager em ;

   //@ public invariant !(tents == 2 && sess == ACTIVE);
   //@ public invariant !(sess == ACTIVE && asess == ACTIVE);

   /*@ requires id >= 1 && mdp.length() >= 6 && mdp.length() <= 10
       && sess == INACTIVE && asess == inactive ; @*/
   //@ ensures sess == ACTIVE && asess == inactive
   public Membre login(int id, String mdp) throws EchecConnexionExp {
      if(tents <= 2 && tents >= 0){
         tents = tents - 1 ;
         try {
            Membre connecte = (Membre) em.createNamedQuery("identifierMembre").
                           setParameter("identificateur",id).
                           setParameter("mot de passe", mdp).getSingleResult();
            sess = ACTIVE ;
            return connecte ;
         } catch(Exception e) {
             throw new EchecConnexionExp("Membre inconnu !!");
         }
       }else{//@assert tents == -1;
         throw new EchecConnexionExp("Vous avez déjà effectué vos deux
                                    tentatives de connexion !!");
      }
    }

    /*@ requires mdp.length() >= 6 && mdp.length() <= 10
       && sess == INACTIVE && asess == inactive ; @*/
    //@ ensures asess == ACTIVE && sess == inactive
    public Membre su(String mdp) throws EchecConnexionExp {
         try {
            Membre connecte = (Membre) em.createNamedQuery("identifierAdmin").
                           setParameter("mot de passe", mdp).getSingleResult();
```

```
            asess = ACTIVE ;
            return connecte ;
         } catch(Exception e) {
             throw new EchecConnexionExp("Admin : echec de connexion !!");
         }
    }
}
```

Le composant Services peut gérer les connexions des administrateurs et des clients. Par contre il ne peut pas les gérer en même temps car c'est un composant EJB de type *stateful*. Nous rappelons que les *stateful beans* partagent les sessions avec un client et garde l'état de communication avec lui mais il ne peut pas être connecté à plusieurs clients en même temps (dans notre cas les composants Client et Admin). Le composant est dupliqué en plusieurs instances et chaque instance communique avec un client particulier.

Le composant autorise au plus deux tentatives de connexion pour les clients ordinaires. Par contre, il ne met aucune restriction pour les administrateurs. Si le composant trouve le composant *entity* qui représente le client membre dans le conteneur (`EntityManager em`) des composants *entity* persistants, la connexion termine avec succès. Sinon, une exception est détectée.

Le lecteur peut de même déduire que les invariants sont préservés après l'appel de l'une des méthodes *login* ou *su*. Conformément aux caractéristiques des composants *stateful*, les sessions du client et de l'administrateur ne peuvent pas être activées en même temps `!(sess == ACTIVE && asess == ACTIVE)`.

```
//@ requires sess == ACTIVE || asess == ACTIVE ;
//@ ensures sess == INACTIVE && asess == INACTIVE;
public Membre deconnexion() {
   if(sess == ACTIVE && asess == INACTIVE){
      sess = INACTIVE ;
      tents = 2 ;
   } else //@assert sess == INACTIVE && asess == ACTIVE
      asess = INACTIVE ;
   return null ;
}
```

La méthode `deconnexion` préserve aussi les invariants et assure la désactivation de la session de l'administrateur ou du client. Les actions *login* et *deconnexion* sont les seules qui modifient l'attribut `sess` dans les deux composants Client et Services. Ceci assure, que durant la communication du client avec le composant Services, l'invariant `!(tents == 2 && sess == ACTIVE)` est préservé et les deux composants ne peuvent pas être situés dans un état où l'invariant est satisfait dans l'un des deux composants et pas dans l'autre. Cette propriété reflète le résultat établi par le théorème 6.4 qui montre que seulement les invariants établis sur les variables totalement contrôlées et les variables locales sont certainement préservés après la composition.

Dans l'implémentation de la méthode `validerLocation`, nous pouvons remarquer que, si le crédit du client est suffisant, une durée de location est affectée pour la location est le panier redevient vide (`panier = VIDE`) comme c'est indiqué dans le tableau 6.2. Les méthodes `ajouterAuPanier` et `validerLocation` sont définies comme suit dans le composant Services.

```
/*@ requires idRes != 0
```

```
        && sess == ACTIVE ; @*/
   //@ ensures panier == NONVIDE
   public void ajouterAuPanier(int idRes) {
      if(panier == VIDE)
         panier = NONVIDE ;
      ...
   }

   /*@ requires idMb != 0 && idLoc != 0 && sess == ACTIVE
       && panier == NONVIDE ; @*/
   //@ ensures panier == VIDE ;
   public boolean validerLocation(int idMb, int idLoc) {
      try{
         LocationRessource loc = (LocationRessource)
                              em.createNamedQuery("identifierLocation").
                              setParameter("id du membre",idMb).
                              setParameter("id de la location",idLoc).
                              getSingleResult();
         int nvcrd = connecte.debiter(loc.getPrix()) ;
         Date debut = new Date() ; Date fin = ... ;
         //@assert debut.getTime() < fin.getTime();
         loc.dureeLocation(debut,fin);
         panier = VIDE ;
         return true ; // action de sortie locValide!
      } catch (CreditInsuffisantExp e) {
          throw new LocNonValideExp("La location ne peut pas être
                                     validée : crédit insuffisant !!")
      }
}}
```

6.6 Synthèse

Dans ce chapitre, nous nous somme intéressés à la vérification, d'une part, de l'interopérabilité des composants en utilisant simultanément les actions les variables partagées, et, d'autre part, de la sûreté du fonctionnement du système après l'opération d'assemblage des composants. Le formalisme des automates d'interface sémantiques proposé permet la vérification de l'interopérabilité entre les composants aux niveaux signature, sémantique et protocole. Les propriétés de sûreté sont des invariants définis sur les variables. Les propriétés d'invariance sont évaluées sur tous les états du système de transitions étiquetées. En particulier, nous étudions la préservation des invariants par la composition et le raffinement.

Nous avons montré également la possibilité d'utiliser les automates d'interface sémantiques pour générer une partie des codes des composants.

Troisième partie

Outillage

Chapitre 7

Implémentations et vérification des automates d'interface sémantiques

Dans ce chapitre, nous présentons un environnement de spécification basé sur le langage *ISIA* (*Language for describing and checking Semantical Interface Automata*) permettant d'implanter le formalisme des automates d'interface sémantiques (cf. chapitre 6). Les automates d'interface sémantiques sont dotés d'une sémantique d'interopérabilité plus riche que celle des automates d'interface classiques. La description de la sémantique des actions est basée sur l'utilisation des paramètres et des variables. Ces variables peuvent être partagées entre la spécification d'un composant et celle de son environnement et permettent de refléter la conduite du comportement du composant et de sa communication avec l'environnement.

L'outil permet de construire la composition des automates d'interface sémantiques compatibles. Le cœur du langage vérifie la compatibilité entre les spécification des automates d'interface sémantiques en vérifiant la compatibilité sémantique des actions partagées. La vérification de la compatibilité sémantique (cf. section 4.2) est effectuée en utilisant le solveur automatique de satisfiabilité CVC3. Les spécifications des automates d'interface sémantiques sont translatées à des spécifications TLA$^+$ pour vérifier leurs sûretés par des techniques de *model-checking*.

Dans la section 7.1, nous présentons brièvement environnement de spécification *ISIA*. Dans la section 7.2, nous présentons comment l'outil prend en charge la vérification de la compatibilité des automates d'interface sémantiques aux niveaux signature, sémantique et protocole. Dans la section 7.3, nous présentons le langage de spécification TLA$^+$ utilisé pour spécifier et vérifier les propriétés de sûreté. Nous présentons aussi une implémentation de l'étude de cas présentée dans le chapitre 6.

7.1 Langage de spécification *ISIA*

L'outil *ISIA* (*Language for describing and checking Semantical Interface Automata*) permet de spécifier et de vérifier des automates d'interface sémantiques. Il est implémenté en Objective Caml [LDF$^+$10]. Une spécification *ISIA* est composée de trois parties :

1. les déclarations des types énumérés et les enregistrements ;
2. la déclaration des variables partagées ;
3. les spécifications des SIAs d'un ensemble de composants primitifs [1].

1. un composant primitif est un composant qui ne comporte pas des sous composants (cf. section 1.6).

7.1.1 Types des variables

Les types supportés par le langage sont les booléens, les entiers, les réels, le type chaîne de caractères, les types énumérés et les enregistrements. Le type booléen est identifié par `bool`, les entiers sont identifiés par le mot clé `int`, les réels sont identifiées par le mot clé `real` et les chaînes de caractères sont identifiées par le mot clé `string`. Les types énumérés et les enregistrements sont déclarés de la manière suivante :

```
enum type_enum {‘valeur_1, ‘valeur_2, ..., ‘valuer_n}

record type_enreg {
   champ_1 : type_1;
   champ_2 : type_2;
   ...
}
```

où `valuer_i` représente la $i^{\text{ème}}$ valeur de type énumératif `type_enum` et *champ_i* représente le $i^{\text{ème}}$ champ de l'enregistrement `type_enreg`.

7.1.2 Variables partagées

Le langage spécifie les automates d'interface sémantiques d'un ensemble de composants $\mathcal{C}$. En se basant sur la définition formelle des automates d'interface sémantiques (cf. sections 6.1), nous définissons la fonction $\Delta : V_{\mathcal{C}} \rightarrow 2^{\mathcal{C}}$ qui associe à chaque variable $v \in V_{\mathcal{C}}$, un sous-ensemble de composants de $\mathcal{C}$ qui partagent et modifient v. Pour chaque variable v dans $V_{\mathcal{C}}$,

$$\Delta(v) = \{c \in \mathcal{C} \mid v \in \bigcup_{a \in \Sigma_{A_c}} Chg_{A_c}(a)\}.$$

où A_c est l'automate d'interface sémantique d'un composant $c \in \mathcal{C}$. L'ensemble des variables partagées est défini de la manière suivante

```
  shared var_1 : type_1 by c_1, ..., c_k ;
         var_2 : type_2 by d_1, ..., d_l ;
         ...
```

où `c_1, ..., c_k` sont les composants qui partagent la variable `var_1` et `d_1, ..., d_l` sont les composants qui partagent la variable `var_2` pour $k,l \geq 1$.

7.1.3 Spécification des automates d'interface sémantiques

La spécification d'un automate d'interface sémantique qui représente un composant primitif est définie de la manière suivante :

```
primitive ComposantPrimitif {
   // Déclaration des variables locales
   local loc_1 : type_1
         loc_2 : type_2

   // Ensemble des états
```

```
   states "s1", "s2", ...
   initial "s1"

   init [condition initiale sur les variables]

   // Ensemble des actions d'entrée, de sortie et internes.
   input ia_1, ia_2, ...
   output oa_1, oa_2, ...
   hidden ha_1, ha_2, ...

   inv [invariant]

   transitions
        // liste des transitions étiquetées par ia
        input ia_1 "s1" -> "s2" "s2" -> "s3"
        in: pi1 : type_pi1, pi2 : type_pi2, ... ; out: po1 : type_po1, ...
        // la sémantique de l'action ia
        pre [précondition de ia]
        transpred [prédicat de transition de ia]
        post [post-condition de ia]

        output oa_1 "s2" -> "s1"
        in: qi1 : type_qi1, qi2 : type_qi2, ... ; out: qo1 : type_qo1, ...
        pre [précondition de oa]
        transpred [prédicat de transition de oa]
        post [post-condition de oa]
        ...
}
```

Les variables locales sont visibles qu'à l'intérieur de la spécification d'interface. La clause `initial` définit l'état initial de l'automate. La clause `init` représente la condition initiale sur les variables. Les clauses `input`, `output` et `hidden` représentent respectivement les ensembles des actions d'entrée, de sortie et internes. La clause `inv` représente l'invariant qui doit être satisfait par la spécification. Pour chaque action a, les paramètres d'entrée et de sortie de a (précédés respectivement par les mots clés `in` et `out`), et la sémantique (pré et post-conditions et prédicat de transition) sont définis sous la clause `transitions`.

Exemple 7.1. Nous allons prendre comme exemple l'automate d'interface sémantique A_C du composant Client présenté dans l'exemple 6.1. Les types prédéfinis, les enregistrements et les variables partagées sont définies de la manière suivante :

```
enum SessionStatut {‘active,‘inactive}
enum PanierStatut {‘vide,‘nonvide}

record Chaine {
   contenu : string ;
   taille : int
}
```

```
record Membre {
   id : int ;
   ...
}

shared sess : SessionStatut by Client, Services ;
       panier : PanierStatut by Client, Services, Location
```

où `SessionStatut` et `PanierStatut` sont des types énumérés qui représentent respectivement l'état de la session du client et l'état du panier d'achat virtuel. Le code suivant représente une partie de la spécification *ISIA* de l'automate A_C est défini comme suit :

```
primitive Client {
    local // pas de variables locales

    states "1", "2", "3", "4", "5"
    initial "1"

    init session = ‘inactive /\ panier = ‘vide

    input connecte, echec, deconnexion, locValide, locNonValide
    output login, ajouterAuPanier, validerLoc

    inv true

    transitions
     output login "1" -> "2"
          in: id : int, mdp : Chaine ; out : membre : Membre
          pre id > 0 /\ mdp.taille <= 10 /\ mdp.taille >= 8 /\ sess = ‘inactive
          transpred sess = ‘inactive /\ unchanged sess,panier
          post membre.id = id

     input connecte "2" -> "3"
          pre sess = ‘inactive
          transpred    sess = ‘inactive
                    /\ sess’ = ‘active
                    /\ unchanged panier
          post sess = ‘active
     ...
}
```

Le prédicat de transition de l'action *login* dans la spécification de l'automate d'interface sémantique A_S peut être exprimé à l'aide d'une structure conditionnelle `if/then/else` (cf. tableau 6.2). ■

```
     output login "1" -> "2"
          ...
          transpred    sess = ‘inactive
```

```
                    /\ if tents >= 0 then
                           tents' = tents - 1
                       else
                           unchanged tents
                    /\ unchanged sess,panier
          ...
}
```

7.2 Architecture de l'outil

L'architecture de l'outil permet de vérifier la compatibilité entre les spécifications des automates d'interface sémantiques conformément à notre approche proposée dans le chapitre 6. La structure de l'outil est présentée dans la figure 7.1.

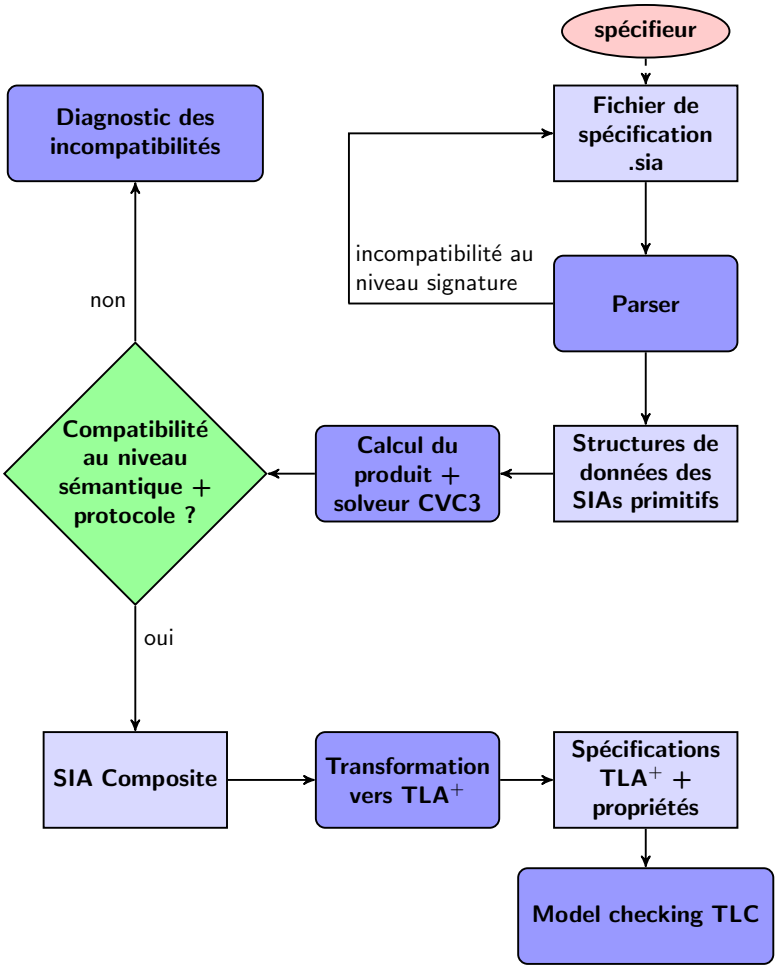


FIGURE 7.1 – Structure de l'outil *ISIA*

Pour chaque deux automates d'interface sémantiques A_1 et A_2 composables, l'outil permet de générer des structures de données qui représentent les deux automates et de vérifier leur compatibilité au niveau signature, protocole et sémantique. L'outil permet aussi de générer des

spécifications TLA$^+$ (cf. section 7.3) pour vérifier les propriétés des automates d'interface sémantiques par des techniques de *model-checking*.

Solveur CVC3

Nous avons choisi d'utiliser le solveur SMT (*Satisfiability Modulo Theories*) CVC3 [BT07, SBD02] pour vérifier la compatibilité sémantique entre les actions partagées des automates d'interface sémantiques. CVC3 est un solveur de satisfiabilité automatique pour plusieurs logiques de premier ordre typées modulo des théories contextuelles (comme l'arithmétique des entiers et les réels, etc.). Brièvement, le solveur CVC3 vérifie si une formule donnée ϕ est valide ou non dans une théorie définie T sous un ensemble d'hypothèses Γ. En d'autres termes, il vérifie si $\Gamma \models_T \phi$ (si ϕ est une conséquence logique de l'union de la théorie T et l'ensemble des hypothèses Γ).

7.3 Vérification des propriétés de sûreté

Dans cette section, nous allons montrer comment l'outil *ISIA* permet de vérifier la sûreté des automates d'interface sémantiques en se basant sur le langage de spécification TLA$^+$. Comme nous venons de mentionner dans la section précédente, l'outil peut générer pour chaque automate d'interface sémantique (primitif ou composite) une spécification TLA$^+$ qui permet de raisonner sur ses propriétés de sûreté et de vivacité décrites en fonction des variables. Dans le cadre de nos travaux, nous avons traité uniquement la préservation des propriétés de sûreté (les invariants) par la composition et le raffinement. Dans l'exemple présenté dans cette section, nous allons montrer comment les deux types de propriétés sont spécifiées en TLA$^+$.

7.3.1 Langage de spécification TLA$^+$

La logique temporelle des actions (*Temporal Logic of Actions*) TLA a été introduite par Leslie Lamport [Lam94] en 1994, TLA permet de spécifier et de raisonner au sujet des systèmes concurrents. Un système et ses propriétés sont représentés dans la même logique. Leslie Lamport a développé progressivement un langage de spécifications, appelé TLA$^+$ étendant TLA par des notations de la théorie des ensembles et une structuration sous forme de modules.

Définition 7.1 (*Logique temporelle des actions*)**.** *La logique temporelle des actions permet de combiner le système de transitions d'un système et ces propriétés sous la forme des formules temporelles. Elle offre une abstraction mathématique sur laquelle se base la description des systèmes réactifs et distribués pour concevoir structurellement des systèmes complexes.*

TLA$^+$ définit deux niveaux de syntaxes [Lam02, Lam94] : les formules d'actions, représentant les états et le système de transitions, et les *formules temporelles* établissant les exécutions et les propriétés. Elle inclut une signature de la logique de premier ordre (fonctions et symboles de prédicats). Les variables en TLA$^+$ se divisent en deux parties : les variables *rigides* relatives aux valeurs, comme dans la logique de premier ordre, et les variables *flexibles* représentant les composants des états.

Les formules d'action sont évaluées entre paires d'états en se basant sur les variables flexibles. Dans cette perspective, deux types sont distingués : les variables flexibles *primées* (représentant les valeurs des variables après l'exécution de l'action) et les variables flexibles *non primées* (représentant les valeurs des variables avant l'action) [2]. Les formules d'action sans variables libres primées sont appelées *formules d'état*.

2. exemples : $hr \in (0..23)$, $hr' = hr + 1$, $n + m' < 3 \times k$, ...

Le comportement du système est spécifié par la formule temporelle suivante :

$$Init \wedge \Box[Next]_{vars} \wedge Liveness$$

où *Init* indique le prédicat initial. Le prédicat *Next* désigne la formule d'actions décrivant les transitions possibles, (elle est la disjonction d'actions élémentaires A_i). La notation $\Box[Next]_{vars}$ [3] est traduite par le fait que toujours l'action *Next* est exécutée ou on reboucle sur le même état ($vars' = vars$). La dernière partie *Liveness* de la spécification représente les conditions d'équité indispensables pour vérifier les propriétés temporelles.

7.3.2 TLC : Model-checker de TLA$^+$

TLC [Lam02] est un *model-checker* pour les spécifications TLA$^+$. TLC vérifie des propriétés en partant d'une spécification $Init \wedge [Next]_{vars} \wedge Liveness$. Si la spécification ne contient pas des formules temporelles à la fin (de la forme $Init \wedge [Next]_{vars}$), TLC ignore la vérification temporelle.

La façon la plus efficace de trouver des erreurs dans une spécification consiste à essayer de vérifier qu'elle satisfait les propriétés associées. TLC peut vérifier que la spécification satisfait une grande classe de formules temporelles. TLC peut également être exécuté sans lui indiquer de propriétés. Dans ce cas, il ne cherchera que les erreurs de blocage (*deadlock*) ou des erreurs absurdes (*silliness errors*) indiquant la violation de la sémantique de TLA$^+$. Le *model-checker* TLC doit avoir comme entrée un module TLA$^+$ et un fichier de configuration. Le fichier de configuration fournit à TLC les noms des spécifications, les propriétés à vérifier et les valeurs des constantes.

7.3.3 Traduction des spécifications *ISIA* en TLA$^+$

Dans ce paragraphe, nous allons montrer comment traduire les spécifications *ISIA* vers TLA$^+$. Nous allons prendre comme exemple l'automate d'interface sémantique A_S du composant Services présenté dans l'exemple 6.1. Dans l'annexe B, nous présentons les spécifications TLA$^+$ complètes des automates A_C, A_S et A'_S des composants Client et Services (cf. exemple 6.4).

Les variables et leurs types sont déclarées de la manière suivante dans le module TLA$^+$ associée à l'automate d'interface sémantique A_S :

EXTENDS *Naturals*, *Sequences*, *TLC*
CONSTANTS *SessionStatut*, *PanierStatut*
VARIABLES *sess*, *panier*, *tents*
$vars \triangleq \langle sess, tents, panier \rangle$

où *SessionStatut* et *PanierStatut* sont respectivement les types énumératifs des variables *sess* et *panier*. Le tuple *vars* regroupe toutes les variables déclarées dans la spécification. Le prédicat initial *Init* est défini de la manière suivante :

$$\begin{aligned} Init \quad \triangleq \quad & \wedge sess = \text{“inactive”} \\ & \wedge tents = 2 \\ & \wedge panier = \text{“vide”} \end{aligned}$$

Le prédicat de transition de l'action *login* dans la spécification du composant Services

```
output login
```

3. *vars* est un n-uplet de toutes les variables.

```
transpred    sess = 'inactive
          /\ if tents >= 0 then
                tents = tents -1
             else
                unchanged tents
          /\ unchanged sess,panier
```

est traduit de la manière suivante en TLA$^+$:

$$
\begin{array}{lll}
\mathit{login} & \triangleq & \land\ \mathit{sess} = \text{“inactive”} \\
 & & \land\ \text{IF } \mathit{tents} \geq 0 \text{ THEN} \\
 & & \qquad \land\ \mathit{tents}' = \mathit{tents} - 1 \\
 & & \quad \text{ELSE} \\
 & & \qquad \land\ \text{UNCHANGED } \langle \mathit{tents} \rangle \\
 & & \land\ \text{UNCHANGED } \langle \mathit{sess}, \mathit{panier} \rangle
\end{array}
$$

Pareillement, le prédicat de transition de l'action *ajouterAuPanier*

```
input login
   transpred    sess = 'active
             /\ panier = 'nonvide
             /\ if panier = 'vide then
                   panier' = 'nonvide
                else
                   unchanged panier
             /\ unchanged sess,tents
```

est traduit de la manière suivante en TLA$^+$:

$$
\begin{array}{lll}
\mathit{ajouterAuPanier} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
 & & \land\ \mathit{panier} = \text{“vide”} \lor \mathit{panier} = \text{“nonvide”} \\
 & & \land\ \text{IF } \mathit{panier} = \text{“vide”} \text{ THEN} \\
 & & \qquad \land\ \mathit{panier}' = \text{“nonvide”} \\
 & & \quad \text{ELSE} \\
 & & \qquad \land\ \text{UNCHANGED } \langle \mathit{panier} \rangle \\
 & & \land\ \text{UNCHANGED } \langle \mathit{sess}, \mathit{tents} \rangle
\end{array}
$$

Le prédicat UNCHANGED $\langle var_1,...,var_n \rangle$ est équivalent à la conjonction $var'_1 = var_1 \land ... \land var'_n = var_n$. Nous n'allons pas montrer toutes les actions, nous contentons de présenter ces deux actions pour expliquer brièvement comment les actions sont traduites en TLA. Le comportement du module est défini par la formule suivante :

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{Liveness}$$

où *Next* est défini par la disjonction de toutes les actions

$$
\begin{array}{lll}
\mathit{Next} & \triangleq & \lor\ \mathit{login} \\
 & & \lor\ \mathit{connecte} \\
 & & \lor\ \mathit{echec} \\
 & & \lor\ \mathit{deconnexion} \\
 & & \lor\ \mathit{ajouterAuPanier}...
\end{array}
$$

et *Liveness* est définie par la formule suivante

$$\begin{aligned} \mathit{Liveness} \triangleq\ & \mathrm{SF}_{\mathit{vars}}(\mathit{deconnexion}) \wedge \mathrm{SF}_{\mathit{vars}}(\mathit{connecte}) \wedge \\ & \mathrm{SF}_{\mathit{vars}}(\mathit{echec}) \wedge \mathrm{SF}_{\mathit{vars}}(\mathit{locValide}) \wedge \\ & \mathrm{SF}_{\mathit{vars}}(\mathit{locNonValide}) \wedge \mathrm{SF}_{\mathit{vars}}(\mathit{crdInsuff}) \wedge \\ & \mathrm{SF}_{\mathit{vars}}(\mathit{nvCredit}) \wedge \mathrm{SF}_{\mathit{vars}}(\mathit{debiter}) \end{aligned}$$

où $\mathrm{SF}_{vars}(a)$ est la condition d'équité forte (*strong fairness*) appliquée à l'action a. La condition $\mathrm{SF}_{vars}(a)$ assure que si l'événement $\mathit{Next} \wedge (\mathit{vars}' \neq \mathit{vars})$ [4] est infiniment souvent activable alors, il peut se produire infiniment souvent. Nous assumons que les actions *deconnexion*, *connecte*, *locValide*, *locNonValide*, *crdInsuff*, *nvCredit*, *debiter* et *echec* sont soumises à des conditions d'équité fortes. Pour plus de détails sur les conditions d'équité, nous invitons le lecteur à consulter le manuel de TLA$^+$ [Lam02].

L'invariant du module est défini par la formule $\neg(\mathit{tents} = 2 \wedge \mathit{sess} = \text{“active”})$. Les conditions d'équité permettent aussi d'assurer la vérification de la formule temporelle $(\mathit{sess} = \text{“active”} \leadsto \mathit{sess} = \text{“inactive”} \equiv \Box(\mathit{sess} = \text{“active”} \Rightarrow \Diamond(\mathit{sess} = \text{“inactive”}))$: si la session du client est active donc elle doit être désactivée après un temps donné. La structure globale du module est définie comme suit

MODULE *Services*

EXTENDS *Naturals*, *Sequences*, *TLC*
CONSTANTS *SessionStatut*, *PanierStatut*
VARIABLES *tents*, *sess*, *panier*

$\mathit{vars} \triangleq \langle \mathit{tents}, \mathit{sess}, \mathit{panier} \rangle$

$$\begin{aligned} \mathit{Init} \triangleq\ & \wedge \mathit{sess} = \text{“inactive”} \\ & \wedge \mathit{tents} = 2 \\ & \wedge \mathit{panier} = \text{“vide”} \end{aligned}$$

/ Déclaration des actions et du prédicat Next */*

/ Déclaration des conditions d'équité Liveness */*

$\mathit{Spec} \triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{vars}} \wedge \mathit{Liveness}$

$\mathit{Invariant} \triangleq \neg(\mathit{tents} = 2 \wedge \mathit{sess} = \text{“active”})$

$\mathit{Live} \triangleq \mathit{sess} = \text{“active”} \leadsto \mathit{sess} = \text{“inactive”}$

La spécification TLA$^+$ de l'automate d'interface sémantique A'_S du composant Services (cf. section B.3) satisfait les deux invariants $\neg(\mathit{tents} = 2 \wedge \mathit{sess} = \text{“active”})$ et $\neg(\mathit{sess} = \text{“active”} \wedge \mathit{sess} = \text{“inactive”})$ et les deux propriétés de vivacité $(\mathit{sess} = \text{“active”} \leadsto \mathit{sess} = \text{“inactive”})$ et $(\mathit{asess} = \text{“active”} \leadsto \mathit{asess} = \text{“inactive”})$.

4. Les actions qui bouclent sur le même état ($\mathit{vars}' = \mathit{vars}$) ne sont pas autorisées.

L'outil permet de générer la composition des automates d'interface sémantique compatibles et de les traduire vers des spécification TLA^+ pour vérifier la préservation des propriétés d'invariance par la composition. Le traitement du raffinement reste une perspective de l'outil.

7.4 Synthèse

Dans ce chapitre, nous avons présenté le langage et l'outil de spécification *ISIA* (*Language for describing and checking Semantical Interface Automata*) qui permettent d'implémenter le formalisme des automates d'interface sémantiques. L'analyseur syntaxique permet de vérifier la compatibilité des actions au niveau signature. L'outil permet de tester la compatibilité des automates d'interface sémantiques et de construire leur composition. Le moteur du langage vérifie la compatibilité sémantique des actions durant le processus d'assemblage en utilisant le solveur automatique de satisfiabilité CVC3. Les spécifications des automates d'interface sémantiques sont translatées à des spécification TLA^+ pour vérifier leurs sûretés par le *model-checker* TLC. Plusieurs exemples de systèmes à base de composants ont été traités pour valider notre approche.

Conclusion et perspectives

1 Conclusion

Concevoir des systèmes sûrs est une exigence essentielle qui doit être prise en compte lors de développement des systèmes informatiques de grande taille. Cependant, la tâche de vérification de la correction de ces systèmes par rapport à leurs spécifications est plus difficile à mettre en œuvre. La violation de certaines propriétés, essentielles à la survie d'un système, pourrait avoir un coût exorbitant.

Dans le cadre de cette thèse, nous sommes intéressés par la spécification et la vérification des systèmes à base de composants logiciels. La conception de tels systèmes par assemblage de composants réutilisables vise essentiellement à réduire leur coût de développement et de maintenance. Un composant logiciel est décrit par la spécification contractuelle de ses interfaces de services fournis et requis. Cette spécification doit, d'une part, ne pas dévoiler les détails de l'implémentation du composant et, d'autre part, décrire suffisamment le composant pour pouvoir l'utiliser. Souvent, les informations fournies par les interfaces des composants ne sont pas suffisantes pour concevoir des systèmes corrects.

De manière générale, les modèles de composants utilisés dans le milieu industriel n'assurent la vérification de l'interopérabilité qu'au niveau des signatures des opérations. Cependant, l'interopérabilité fonctionnelle de deux ou plusieurs composants doit être vérifiée aux niveaux des signatures et sémantiques des opérations, et de leurs protocoles comportementaux. Cette vérification doit être effectuée en deux phases : lors du processus d'assemblage des composants en vérifiant la compatibilité et, ensuite, la sûreté de fonctionnement des composants. Plus précisément, durant le processus d'assemblage, le concepteur du système doit s'assurer que les composants sont interopérables et ne provoquent pas d'incompatibilités. Après le processus d'assemblage, le système doit fonctionner correctement.

Pour augmenter le degré de confiance des systèmes à base de composants, nous avons proposé une approche formelle basée sur le modèle des automates d'interface pour vérifier l'assemblage. Les formalismes proposés permettent de décrire les niveaux signature, sémantique et protocole.

Le formalisme des automates d'interface [dAH01, AH05] permet de décrire les protocoles des composants. Il est soumis à une approche *optimiste* de composition et de raffinement. Deux composants sont compatibles s'il existe un environnement qui empêche leur composition de provoquer des situations de blocage. Le raffinement des automates d'interface est défini également en respectant les contraintes de l'environnement.

Nous avons proposé, dans un premier temps, un modèle d'interface basé sur les automates d'interface en prenant en compte la sémantique des paramètres des actions. Cette sémantique

est décrite par des pré et post-conditions sur les paramètres. L'interopérabilité est vérifiée en calculant la satisfiabilité d'implications entre les pré et post-conditions basées sur les règles de sous-typage sémantique proposées dans [Ame91].

Dans le cas où l'interopérabilité implicite des composants n'est pas envisageable, nous proposons une méthodologie d'adaptation des composants dont les contrats sont décrits par des automates d'interface enrichis par la sémantique des actions. Premièrement, cette méthodologie permet de vérifier l'adaptabilité sémantique entre les actions non partagées de deux automates d'interface qui sont censées être impliquées dans une communication. Deuxièmement, elle permet d'exhiber un automate d'interface adaptateur qui garantit leur communication. L'algorithme proposé teste l'adaptabilité de deux automates d'interface et il génère un adaptateur s'ils sont adaptables.

Pour vérifier l'interopérabilité des composants qui communiquent par des actions et des variables partagées, nous avons rendu plus expressif le modèle des automates d'interface par l'ajout des variables. Le nouveau formalisme, appelé automates d'interface sémantique, utilise les actions et les variables pour vérifier, d'une part, l'interopérabilité et, d'autre part, pour raisonner sur le comportement des composants en vérifiant leurs propriétés de sûreté. Nous avons étudié en particulier la préservation d'invariants par la composition et par le raffinement. De plus, nous avons montré également comment utiliser les automates d'interface sémantiques pour générer une partie de l'implémentation des composants.

Pour valider nos travaux, nous avons présenté un environnement de spécification permettant de décrire les automates d'interface sémantiques des composants. Cet outil, appelé *ISIA* (*Language for describing and checking Semantical Interface Automata*), permet de vérifier la compatibilité entre les automates d'interface sémantiques, les composer, et vérifier leur sûreté par des techniques de *model-checking*. La compatibilité sémantique des actions est vérifiée en utilisant le solveur de satisfiabilité CVC3. L'outil permet également de traduire les spécifications des automates d'interface sémantiques vers des modules TLA$^+$ vérifiables par le *model-checker* TLC.

2 Perspectives

Nous traçons les perspectives suivantes pour poursuivre nos travaux :

- la prise en compte des aspects non-fonctionnels ;
- l'extension de l'approche d'adaptation proposée dans le chapitre 5 par l'utilisation des variables ;
- l'étude de la préservation et la dérivation des propriétés temporelles des composants ;
- étendre l'outil pour supporter les spécifications architecturales et la vérification du raffinement.

La première perspective consiste à adapter les formalismes proposés dans la thèse pour vérifier les exigences de performance et de qualité, appelées couramment les propriétés non-fonctionnelles. Ces propriétés sont requises pour s'assurer du bon fonctionnement d'un système à base de composants dans un environnement opérationnel et réactif. Elles recouvrent les disponibilités des composants, leurs temps de réponse, etc. Ces propositions doivent s'inspirer des travaux comme ceux proposés dans [AHS02] qui traitent les contraintes temporisées des automates d'interface

pour vérifier l'interopérabilité et l'adaptation des composants.

La deuxième consiste à proposer une approche d'adaptation des automates d'interface sémantiques. La méthodologie d'adaptation proposée, dans le chapitre 5, ne supporte pas l'utilisation des variables. On pourrait utiliser ces variables pour générer le code du composant adaptateur et de vérifier la dérivation des propriétés de sûreté.

La troisième piste de travail revêt un interêt industriel. Elle consiste à étudier la préservation et la déduction des propriétés temporelles des systèmes à base de composants. Dans la partie outillage, nous avons montré uniquement la possibilité de vérifier les propriétés temporelles au niveau des spécification TLA^+, il reste à définir les bases théoriques de ce problème et le traiter plus formellement comme nous l'avons fait pour les propriétés d'invariance.

La quatrième perspective consiste à étendre l'outil *ISIA* pour supporter le raffinement des automates d'interface sémantiques et l'architecture des composants en spécifiant les connexions entre eux.

Bibliographie

[AAA07] Pascal André, Gilles Ardourel, and Christian Attiogbé. Protocoles d'utilisation de composants, Spécification et analyse en Kmelia. In *13e Conférence Francophone sur les Langages et Modèles à Objets*, pages 19–34. Hermès Sciences Publications - Lavoisier, 2007.

[AAS+06] B. Thomas Adler, Luca De Alfaro, Ro Dias da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc, a tool for interface compatibility and composition. In *The Proceedings 18th International Conference on Computer Aided Verification (CAV), volume 4144 of Lecture Notes in Computer Science*, pages 59–62. Springer, 2006.

[Abr96] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[ACC02] © Projet ACCORD. Assemblage de composants par contrats, état de l'art et de la standardisation. livrable 1.1 - 1, Juin 2002.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.

[AEB08] Xabier Aretxandieta, Xabier Elkorobarrutia, and Franck Barbier. Component adaptation for correctness in composite systems. In *Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*, pages 130–137, Washington, DC, USA, 2008. IEEE Computer Society.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6 :213–249, July 1997.

[AH01] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *EMSOFT '01 : Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[AH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *Engineering Theories of Software-intensive Systems*, NATO Science Series : Mathematics, Physics, and Chemistry 195, pages 83–104. Springer, 2005.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *The Proc. of the 9th Int. Conf. on Concurrency Theory*, pages 163–178, London, UK, 1998. Springer-Verlag.

[AHS02] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, pages 108–122, London, UK, 2002. Springer-Verlag.

[AL03] Paul C. Attie and David H. Lorenz. Correctness of model-based component composition without state explosion. In *In : ECOOP 2003 Workshop on Correctness of Modelbased Software Composition*, 2003.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, CMU-CS-97-144 ed., Carnegie Mellon University., 1997.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.

[AP05] J. Adamek and F. Plasil. Component composition errors and update atomicity : static analysis : Research articles. *J. Softw. Maint. Evol.*, 17 :363–377, September 2005.

[AS04] Luca de Alfaro and Mariëlle Stoelinga. Interfaces : A game-theoretic framework for reasoning about component-based systems. *Proc. of FOCLASA 2003. ENTCS*, 97 :3–23, July 2004.

[ASF+05] Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable interfaces. In *The Proc. 5th Int. WS. on Frontiers of Combining Systems, LNAI 3717*, pages 81–105. Springer-Verlag, 2005.

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74 :45–54, January 2005.

[BBČ+08] Nikola Beneš, Luboš Brim, Ivana Černá, Jiří Sochor, Pavlína Vařeková, and Barbora Zimmerova. The CoIn Tool : Modelling and Verification of Interactions in Component-Based Systems. In *Proceedings of the Workshop on Formal Aspects of Component Software (FACS'08)*, pages 221–225, September 2008.

[BBPR08] Jiřái Barnat, Luboš Brim, Pavel, and Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

[BBv+06] Jiřái Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. Divine - a tool for distributed verification. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the 7th Int. Symp. CBSE 2004, Edinburgh, UK*, Lecture Notes in Computer Science, pages 7–22. Springer, 2004.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36 :1257–1284, September 2006.

[BCM03] Françoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *in Inter. Symp. on Distributed Objects and Applications (DOA*, pages 1226–1242. Springer-Verlag, 2003.

[Ben87] John K. Bennett. The design and implementation of distributed smalltalk. *SIGPLAN Not.*, 22 :318–330, December 1987.

[Ben88] John Knov Bennett. *Distributed smalltalk : inheritance and reactiveness in distributed systems*. PhD thesis, Seattle, WA, USA, 1988. Order No : GAX88-10907.

[BFG+93] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 188–191, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[BHB08] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Dynamic adaptive software components : the mocas approach. In *Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, CSTST '08, pages 517–524, New York, NY, USA, 2008. ACM.

[BM03] Antonia Bertolino and Raffaela Mirandola. Modeling and analysis of non-functional properties in component-based systems. *The proceedings of Int. Wshop. on Test and Analysis of Component-Based Systems (TACoS'03 of ETAPS'03), ENTCS*, 82(6) :158 – 168, 2003.

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, Beijing, 5.0 edition, 2006.

[Bos00] Jan Bosch. *Design and use of software architectures : adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[Bru03] Eric Bruneton. Julia tutorial. Technical Report Version 2.0, France Telecom R&D, Novembre 2003.

[BS00] Rémi Bastide and Ousmane Sy. Towards components that plug and play. In *Proceedings of the ECOOP 2000 Workshop on Object Interoperability (WOI'00)*, pages 3–12, June 2000.

[BSP99] Rémi Bastide, Ousmane Sy, and Philippe A. Palanque. Formal specification and prototyping of corba systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 474–494, London, UK, 1999. Springer-Verlag.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

[BW97] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical report, Workshop on Foundations of Component-Based Systems, Zürich. TR No. 122, Turku Centre for Computer Science, Turku, Finland, 1997.

[Bü66] J. Richard Büchi. Symposium on decision problems : On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science, Proceeding of the 1960 International Congress*, volume 44, pages 1–11. Elsevier, 1966.

[CD00] John Cheesman and John Daniels. *UML components : a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CDH+10] Samir Chouali, Julien Dormoy, Ahmed Hammad, Jean-Michel Hufflen, Sebti Mouelhi, Olga Kouchnarenko, Hassan Mountassir, Bruno Tatibouët, et al. Assemblage des composants digne de confiance : de l'ingénierie des besoins aux spécifications formelles. *Génie Logiciel*, 95 :13–18, décembre 2010.

[CFTV00] C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending corba interfaces with π-calculus for protocol compatibility. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 208–225, Washington, DC, USA, 2000. IEEE Computer Society.

[CKV98] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in java. In *in Java. Concurrency Practice and Experience*, pages 1043–1061, 1998.

[CL02] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[CL06] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[CMK98] Il-Hyung Cho, John D. McGregor, and Lee Krause. A protocol based approach to specifying interoperability between objects. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, pages 84–, Washington, DC, USA, 1998. IEEE Computer Society.

[CMM10a] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adapting component behaviours using interface automata. In *Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '10, pages 119–122, Washington, DC, USA, 2010. IEEE Computer Society.

[CMM10b] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adapting components using interface automata enriched by action semantics. In *Proceedings of 1st International Conference on Formal Verification of Object-Oriented Software*, pages 7–21, Paris, France, 2010.

[CMM10c] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Assembly of components based on interface automata and UML component model. In *CAL'10, 4e Conf. Francophone sur les Architectures Logicielles*, volume RNTI-L-5 of *RNTI, Revue des Nouvelles Technologies de l'Information*, pages 73–85, Pau, France, mars 2010. Cépaduès éditions.

[CMM10d] Samir Chouali, Hassan Mountassir, and Sebti Mouelhi. An I/O automata-based approach to verify component compatibility : Application to the cycab car. *Electron. Notes Theor. Comput. Sci.*, 238 :3–13, June 2010.

[CMP08] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software adaptation. *Special Issue on Coordination and Adaptation Techniques*, 14(13) :2107–2109, 2008.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[Cou87] Joëlle Coutaz. Pac : an implementation model for dialog design. *Proceedings of the Interact'87 conference*, page 431–436, September 1987.

[CPS06] Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing behavioural mismatch in software composition. In *The Proc. of Inter. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'06)*, volume 4037 of *LNCS*, pages 63–77. Springer-Verlag, 2006.

[CPS08] Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Eng.*, 34 :546–563, July 2008.

[CPT99] Carlos Canal, Ernesto Pimentel, and José M. Troya. Conformance and refinement of behavior in π-calculus, 1999.

[CS03] Inc. Cincom Systems. Distributed smalltalk application developer's guide. Technical report, 55 Merchant Street Cincinnati, Ohio 45246, U.S.A., 2003.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5) :109–120, 2001.

[DBMN08] Marlon Dumas, Boualem Benatallah, and Hamid R. Motahari Nezhad. Web service protocols : Compatibility and adaptation. *IEEE Data Engineering Bulletin*, 31(3) :40–44, 2008.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 258–267, Washington, DC, USA, 1996. IEEE Computer Society.

[DW99] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML : the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[EDG+06] Huáscar Espinoza, Hubert Dubois, Sébastien Gérard, Julio Medina, Dorina Petriu, and Murray Woodside. Annotating uml models with non-functional properties for quantitative analysis. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 79–90. Springer Berlin / Heidelberg, 2006.

[FGK03] Andres Faras, Yann-Gael Gueheneuc, and Alfred Kastler. On the coherence of component protocols. In *In Proc. of the Int. Workshop on Software Composition (SC'03), ENTCS*, volume 82(5). elseiver, 2003.

[Gar95] David Garlan. An introduction to the aesop system, July 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKC99a] Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6 :7–35, January 1999.

[GKC99b] Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engineering*, 6 :7–35, 1999.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme : an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97. IBM Press, 1997.

[GP95] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Trans. Softw. Eng.*, 21 :269–274, April 1995.

[Gro00] The Open Group. Architecture description markup language (adml), 2000. `http://www.opengroup.org/architecture/adml/adml_home.htm`.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[GS05] Jennifer Greene and Andrew Stellman. *Applied software project management.* O'Reilly, first edition, 2005.

[Gud01] Martin Gudgin. *Essential IDL : Interface design for COM.* Addison-Wesley, Essex, UK, 2001.

[HA50] David Hilbert and Wilhelm Ackermann. *Principles of Mathematical Logic (English).* Chelsea, 1950.

[Han99] Jun Han. Semantic and usage packaging for software components. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 7–8, London, UK, 1999. Springer-Verlag.

[Har87] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8 :231–274, June 1987.

[Hei95] Sandra Heiler. Semantic interoperability. *ACM Comput. Surv.*, 27 :271–273, June 1995.

[Hem05] David Hemer. A formal approach to component adaptation and composition. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38*, ACSC '05, pages 259–266, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[HK00] Peter Herrmann and Heiko Krumm. A framework for modeling transfer protocols. *Comput. Netw.*, 34 :317–337, August 2000.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21 :666–677, August 1978.

[Hod01] Wilfrid Hodges. Classical logic I : First-order logic. the blackwell guide to philosophical logic, 2001.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23 :279–295, May 1997.

[Huc] Thomas Huckle. Collection of software bugs. `http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html`.

[IF10] ISO/IEC-FCD-42010. Systems and software engineering – architecture description, Juin 2010.

[Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.). First edition in 1986.* Prentice Hall International, Inc., Upper Saddle River, NJ, USA, 1990.

[KKS10] Prabhu Shankar Kaliappan, Hartmut Kö, and Sebastian Schmerl. Model-driven protocol design based on component oriented modeling. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 613–629, Berlin, Heidelberg, 2010. Springer-Verlag.

[Kof07] Jan Kofroň. *Behavior Protocols Extensions.* PhD thesis, Department of Distributed and Dependable Systems, Charles University of Prague, Czech Republic, 2007.

[Kon95] Dimitri Konstantas. *Interoperation of object-oriented applications*, pages 69–95. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16 :872–923, May 1994.

[Lam02] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML : A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[LC06] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML, 2006. `http://www.eecs.ucf.edu/~leavens/JML/`.

[LDF+10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.12 documentation and user's manual, 2010.

[LM95] Doug Lea and Jos Marlowe. Interface-based protocol specification of open systems using psl. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 374–398, London, UK, 1995. Springer-Verlag.

[LNW06] Kim Larsen, Ulrik Nyman, and Andrzej Wąsowski. Interface input/output automata. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006 : Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2006.

[LPC+08] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual, 2008. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/`.

[LS00] Gary T. Leavens and Murali Sitaraman, editors. *Foundations of component-based systems*. Cambridge University Press, New York, NY, USA, 2000.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC '87 : Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151, New York, NY, USA, 1987. ACM.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16 :1811–1841, November 1994.

[LW01] Barbara H. Liskov and Jeannette M. Wing. *Behavioural subtyping using invariants and constraints*, pages 254–280. Cambridge University Press, New York, NY, USA, 2001.

[MCM09] Sebti Mouelhi, Samir Chouali, and Hassan Mountassir. Refinement of interface automata strengthened by action semantics. *Electron. Notes Theor. Comput. Sci.*, 253 :111–126, October 2009.

[MCM11] Sebti Mouelhi, Samir Chouali, and Hassan Mountassir. Invariant preservation by component composition using semantical interface automata. In *Proceedings of the Sixth International Conference on Software Engineering Advances ICSEA 2011 (IARIA Conferences), to appear*. IEEE Computer Society, 2011.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[Med96] Nenad Medvidovic. Formal modeling of software architectures at multiple levels of abstraction. In *In Proceedings of the California Software Symposium*, pages 28–40, 1996.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction, 2nd edn 1997*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Mey92a] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10) :40–51, 1992.

[Mey92b] Bertrand Meyer. *Eiffel : the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mic] Microsoft. .NET. `http://www.microsoft.com/net/`.

[Mil99] Robin Milner. *Communicating and Mobile Systems : the Pi-Calculus*. Cambridge University Press, June 1999.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency : state models & Java programs*. John Wiley & Sons, Inc., NY, USA, 1999.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[MLS06] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation : Specification and Verification. In *The proceedings of the 11th International Workshop on Component Oriented Programming (WCOP'06)*, pages 23–30, Nantes, France, July 2006.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 377–410, New York, NY, USA, 1990. ACM.

[MPM08] Tarek Melliti, Pascal Poizat, and Sonia Ben Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 146–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[MR98] Lycett Mark and J. Paul Ray. Component-based development : Dealing with non-functional aspects of architecture. In *In ECOOP '98 Workshop on Component-Oriented Programming*, pages 9–8, 1998.

[MRR03] Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault. A Behavioral Model of Component Frameworks. Technical Report RR-5065, ORION - INRIA Sophia Antipolis - INRIA, December 2003.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26 :70–93, January 2000.

[Nie93] Oscar Nierstrasz. Regular types for active objects. *SIGPLAN Not.*, 28 :1–15, October 1993.

[OMG97] OMG. Common object request broker architecture (corba/iiop), 1997.

[OMG05] OMG. Unified modeling language UML infrastructure, version 2.0 ©. Technical report, 2005.

[OMG06] OMG. Corba component model 4.0 specification. Technical Report Version 4.0, April 2006.

[OSG] OSGI. OSGI Service Gateway Specification, Release 1.0. `http://www.osgi.org`.

[Pad09] Luca Padovani. Formal methods for web services. chapter Contract-Based Discovery and Adaptation of Web Services, pages 213–260. Springer-Verlag, Berlin, Heidelberg, 2009.

[PdAHSV02] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis : two faces of the same coin. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 132–139, New York, NY, USA, 2002. ACM.

[PNPR05] Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. Java implementation of a component model with explicit symbolic protocols. In *In Proceedings of the 4th International Workshop on Software Composition (SC'05), volume 3628 of LNCS*, pages 115–124. Springer-Verlag, 2005.

[Pnu97] A. Pnueli. The temporal logic of programs. Technical report, Jerusalem, Israel, Israel, 1997.

[Pre00] Roger S. Pressman. *Software Engineering : A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2000.

[Pri67] A. Prior. *Past, Present and Future*. Oxford University Press, London, 1967.

[Pun99] Franz Puntigam. Non-regular process types. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, pages 1334–1343, London, UK, 1999. Springer-Verlag.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28 :1056–1076, November 2002.

[RDT97] Group Computer Systems Lab Stanford University - Rapide Design Team. Guide to the Rapide 1.0 language. reference manuals. Technical report, July 1997.

[Ree03] Trygve Reenskaug. The model-view-controller (mvc). its past and present, 2003. University of Oslo, Norway.

[SÖ5] Mario Südholt. A model of components with non-regular protocols. In *In Proceedings of the 4th International Workshop on Software Composition (SC'05), LNCS*, volume 3628, pages 99–113. Springer, 2005.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. Cvc : A cooperating validity checker. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 500–504, London, UK, 2002. Springer-Verlag.

[Sch78] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM.

[Sch03] H. Schmidt. Trustworthy components-compositionality and prediction. *J. Syst. Softw.*, 65 :215–225, March 2003.

[SEG08] R. Seguel, R. Eshuis, and P. Grefen. An overview on protocol adaptors for service component integration. Technical report, BETA Working Paper Series WP 265, Eindhoven University of Technology, 2008.

[SG96] Mary Shaw and David Garlan. *Software architecture : perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SG03] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 374–384, Washington, DC, USA, 2003. IEEE Computer Society.

[SKMB90] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of ICCAD'90*, pages 92–95, 1990.

[SM96] Inc. Sun Microsystems. Javabeans, version 1.00, java.sun.com/beans. Technical report, Sun Microsystems, Inc., Sun Microsystems, Inc. Palo Alto, CA, December 1996.

[SM03] Inc. Sun Microsystems. Java^{tm} 2 platform enterprise edition specification, v1.4. Technical report, Sun Microsystems, Inc., Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054, U.S.A., November 2003.

[SP97] Clemens Szyperski and Cuno Pfister. Workshop on component-oriented programming, summary. In *In Mühlhäuser, M. (ed.) Special Issues in Object-Oriented Programming – ECOOP '96 Workshop Reader*, Heidelberg, 1997. dpunkt Verlag.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TB85] P.P. Tanner and W. Buxton. Some issues in future user interface management system development, 1985.

[vVZ07] Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component substitutability via equivalencies of component-interaction automata. *Electron. Notes Theor. Comput. Sci.*, 182 :39–55, June 2007.

[VW01] Wim Vanderperren and Bart Wydaeghe. Towards a new component composition process. *Proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems,*, pages 322 – 329, 2001.

[VZ05] Pavlína Vařeková and Barbora Zimmerova. Component-interaction automata for specification and verification of component interactions. pages 71–75, 2005.

[Weg96] Peter Wegner. Interoperability. *ACM Comput. Surv.*, 28 :285–287, March 1996.

[Wie03] Karl Eugene Wiegers. *Software Requirements*. Microsoft Press, Redmond, WA, USA, 2 edition, 2003.

[Wik11] Wikipédia. Architecture logicielle, Janvier 2011. `http://fr.wikipedia.org/wiki/Architecture_logicielle`.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14 :221–227, April 1971.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, Reading, MA, 1999.

[WSG08] Jules White, Douglas Schmidt, and Aniruddha Gokhale. Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation : a Case Study. *Software and System Modeling*, 7(1) :3–23, 2008.

[You00] Ralph R. Young. *Effective requirements practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19 :292–333, March 1997.

[ZW95] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching : a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4 :146–170, April 1995.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6 :333–369, October 1997.

Annexe A

Preuves des théorèmes 3.1, 3.3 et 4.1

A.1 Preuve du théorème 3.1

Théorème 3.1. *Soient trois automates d'interface A_1, A_2 et A_3 composables mutuellement, alors les automates d'interface $(A_1 \parallel A_2) \parallel A_3$ et $A_1 \parallel (A_2 \parallel A_3)$ sont égaux.*

Démonstration. La preuve du théorème est déduite à partir de la preuve détaillée dans [AH05]. Les automates d'interface A_1, A_2 et A_3 sont mutuellement composables. Considérons le produit $A_1 \otimes A_2 \otimes A_3$, l'associativité de $\otimes$ (preuve dans [CL06]) implique que $(A_1 \otimes A_2) \otimes A_3 = A_1 \otimes (A_2 \otimes A_3) = (A_1 \otimes A_3) \otimes A_2$. Un état (s_1, s_2, s_3) est illégal dans $A_1 \otimes A_2 \otimes A_3$ s'il y a une paire (s_i, s_j) dans le produit $A_i \otimes A_j$ pour i et $j \in \{1, 2, 3\}$ et $i \neq j$. Un état (s_1, s_2, s_3) est incompatible dans $A_1 \otimes A_2 \otimes A_3$ s'il existe un état illégal (s'_1, s'_2, s'_3) dans $A_1 \otimes A_2 \otimes A_3$ atteignable de manière autonome à partir de (s_1, s_2, s_3) (atteignable en activant uniquement les des transitions étiquetées par des actions interne ou de sortie). Pour $k \geq 0$, un état (s_1, s_2, s_3) est incompatible de rang k s'il existe un état illégal (s'_1, s'_2, s'_3) atteignable à partir de (s_1, s_2, s_3) de manière autonome en activant au plus k transitions.

Pour prouver que la composition $A_1 \parallel A_2 \parallel A_3$ est associative, il suffit de prouver que $A_1 \parallel A_2 \parallel A_3$ est le produit associatif $A_1 \otimes A_2 \otimes A_3$ en enlevant tous les états incompatibles. La preuve est divisée en deux étapes. Premièrement, nous montrons qu'un état (s_1, s_2, s_3) est incompatible s'il existe une paire (s_i, s_j) incompatible dans n'importe quel produit $A_i \otimes A_j$ pour i et $j \in \{1, 2, 3\}$ et $i \neq j$. Deuxièmement, nous devons montrer que tout (s_1, s_2, s_3) incompatible dans le produit $A_1 \otimes A_2 \otimes A_3$, il existe un état illégal (s'_1, s'_2, s'_3) atteignable à partir de (s_1, s_2, s_3) de manière autonome. Pour cela, il suffit de montrer que s'il existe certaines transitions étiquetées par des actions d'entrée atteignables à partir de (s_i, s_j) et supprimées en calculant une composition $A_i \parallel A_j$ pour i et $j \in \{1, 2, 3\}$ et $i \neq j$, alors si on considère le produit $A_1 \otimes A_2 \otimes A_3$ en enlevant ces transitions, il y aura toujours une exécution autonome entre (s_1, s_2, s_3) et un état illégal (s'_1, s'_2, s'_3).

1. Soit un état (s_1, s_2, s_3) dans le produit $A_1 \otimes A_2 \otimes A_3$ et une projection (s_i, s_j) de (s_1, s_2, s_3) incompatible de rang k dans le produit $A_i \otimes A_j$, nous montrons que (s_1, s_2, s_3) est incompatible de rang k' tel que $k' \leq k$. Soit le chemin autonome σ le plus petit entre (s_i, s_j) et un état illégal (t_i, t_j) dans le produit $A_i \otimes A_j$. Nous avons trois cas de figure. Premièrement, si (s_i, s_j) est un état illégal dans $A_i \otimes A_j$ (de rang 0), alors (s_1, s_2, s_3) est un état illégal de rang 0 dans le produit $A_1 \otimes A_2 \otimes A_3$. Deuxièmement, si la première transition dans σ correspond à une transition étiquetée par une action interne ou de sortie dans le produit $A_1 \otimes A_2 \otimes A_3$, alors l'état (s'_1, s'_2, s'_3) successeur de (s_1, s_2, s_3), en activant cette transition,

est incompatible de rang k-1. La preuve est itérée par induction jusqu'à l'état illégal (t_i, t_j) de rang 0 qui correspond à un état illégal (t_1, t_2, t_3) de rang 0 dans le produit $A_1 \otimes A_2 \otimes A_3$ (le premier cas). Troisièmement, si la première transition dans σ correspond à une transition étiquetée par une action de sortie dans le produit $A_i \otimes A_j$ qui n'est pas acceptée en entrée dans le produit total $A_1 \otimes A_2 \otimes A_3$, alors (s_1, s_2, s_3) est un état illégal de rang 0 dans $A_1 \otimes A_2 \otimes A_3$. Nous avons donc prouver qu'un état (s_1, s_2, s_3) est incompatible s'il existe une paire (s_i, s_j) incompatible dans n'importe quel produit $A_i \otimes A_j$ pour i et $j \in \{1, 2, 3\}$ et $i \neq j$.

2. Soit un état s incompatible dans le produit $A_1 \otimes A_2 \otimes A_3$. Supposant qu'il existe des transitions étiquetées par des actions d'entrée dans le produit $A_i \otimes A_j$ (pour i et $j \in \{1, 2, 3\}$ et $i \neq j$) atteignables à partir de r (la projection de s dans $A_i \otimes A_j$) et supprimées de $A_i \otimes A_j$, en calculant la composition $A_i \parallel A_j$, et du produit complet $A_1 \otimes A_2 \otimes A_3$. Le seul type de transitions qui peuvent être supprimées de cette manière sont les transitions internes (t, a, t') dans $A_1 \otimes A_2 \otimes A_3$ dont la projection (v, a, v') dans $A_i \otimes A_j$ est une transition étiquetée par une action d'entrée. Cette transition est synchronisée par son équivalente de sortie dans le produit complet. Une fois (t, a, t') est retirée, l'action d'entrée a n'est plus activable à partir de l'état v parce que les automates d'interface sont déterministes. Par conséquent, dans le produit final, l'état t est un état illégal. Ainsi, même après la suppression de la transition (t, a, t') du produit $A_1 \otimes A_2 \otimes A_3$, il y a un chemin autonome entre s et un état illégal, par exemple t.

Nous pouvons donc déduire que la composition des automates d'interface composables est associative. □

A.2 Preuve du théorème 3.3

Théorème 3.3. *Soient trois automates d'interface A, A' et E tel que A' et E sont composables et $\Sigma^{\mathrm{I}}_{A'} \cap \Sigma^{\mathrm{O}}_{E} \subseteq \Sigma^{\mathrm{I}}_{A} \cap \Sigma^{\mathrm{O}}_{E}$. Si $A \sim E$ et $A' \preceq A$, alors $A' \sim E$ et $A' \parallel E \preceq A \parallel E$.*

Démonstration. Pour prouver que $A' \sim E$ sous les prémisses du théorème, nous devons montrer que chaque chemin autonome σ' qui mène à un état illégal dans $A' \otimes E$ à partir de l'état initial $i_{A' \otimes E}$ est simulé transition par transition, par un chemin autonome σ qui mène à un état illégal dans $A \otimes E$ à partir de l'état initial $i_{A \otimes E}$. Soit une transition $(r, a, r') \in \sigma'$, nous distinguons trois cas :

- si $a \in \Sigma^{\mathrm{O}}_{A'}$, alors a est une action d'entrée dans E. Selon la définition 3.11, nous avons $\Sigma^{\mathrm{O}}_{A} \supseteq \Sigma^{\mathrm{O}}_{A'}$ et $\Sigma^{\mathrm{O}}_{A} \cap \Sigma^{\mathrm{I}}_{E} \supseteq \Sigma^{\mathrm{O}}_{A'} \cap \Sigma^{\mathrm{I}}_{E}$. Donc, a est aussi une action de sortie dans A. La projection (v, a, v') de (r, a, r') dans A' est simulée par une suite de transitions internes de taille $l \geq 0$ qui commence à partir d'un état w et suivie par une transition (w'', a, w') dans A. Cette suite de transition internes de taille l correspond à une suite de transitions internes de la même taille dans $A \otimes E$ qui commence à partir d'un état $s = (w, e)$ et qui termine par (w'', e) tel que $e \in S_E$. Si a n'est pas activable dans E à partir de e alors nous avons atteint un état illégal dans $A \otimes E$, soit l'état (w'', e). Sinon, il existe une transition $(s'', a, s') \in \delta_{A \otimes E}$ telle que $a \in \Sigma^{\mathrm{H}}_{A \otimes E}$ et $s'' = (w'', e)$ et la simulation continue.
- si $a \in \Sigma^{\mathrm{H}}_{A'}$, la projection (v, a, v') de (r, a, r') dans A' est simulée par une suite de transitions internes de taille $l \geq 0$ qui commence à partir d'un état w dans A. Cette suite de transition internes de taille l correspond à une suite de transitions internes de la même

taille dans $A \otimes E$ qui commence à partir d'un état $s = (w, e)$ et qui termine par (w', e) tel que $e \in S_E$; et la simulation continue.

– si $a \in \Sigma^{\mathrm{I}}_{A'}$, alors a est activable en sortie dans E. Selon la prémisse $\Sigma^{\mathrm{I}}_{A'} \cap \Sigma^{\mathrm{O}}_{E} \subseteq \Sigma^{\mathrm{I}}_{A} \cap \Sigma^{\mathrm{O}}_{E}$, a est aussi une action d'entrée dans A. La projection (v, a, v') de (r, a, r') dans A' est simulée par une transition (e, a, e') dans E. L'état e correspond à un état atteignable $s = (w, e) \in S_{A \otimes E}$. Si a n'est pas activable dans A à partir de w alors nous avons atteint un état illégal dans $A \otimes E$, soit l'état (w, e). Sinon, il existe une transition $(s, a, s') \in \delta_{A \otimes E}$ telle que $a \in \Sigma^{\mathrm{H}}_{A \otimes E}$ et la simulation continue.

Pour prouver que $A' \parallel E \preceq A \parallel E$, il suffit de vérifier que (1) $\Sigma^{\mathrm{I}}_{A \parallel E} \subseteq \Sigma^{\mathrm{I}}_{A' \parallel E}$ et $\Sigma^{\mathrm{O}}_{A \parallel E} \supseteq \Sigma^{\mathrm{O}}_{A' \parallel E}$ (triviale) et (2) qu'il existe une simulation alternée $\preceq'$ entre $A' \parallel E$ et $A \parallel E$ tel que $i' \preceq' i$ $i \in I_{A \parallel E}$ et $i' \in I_{A' \parallel E}$. Soit une simulation alternée $\preceq$ entre A et A' tel que $i_{A'}$ et i_A sont respectivement les états initiaux de A et A'. Une simulation alternée $\preceq'$ entre $A' \parallel E$ et $A \parallel E$ peut être définit comme suit : $(s', t') \preceq' (s, t)$ ssi (1) $s' \preceq s$, (2) $t = t'$ et (3) (s, t) n'est pas un état illégal de $A \otimes E$. □

A.3 Preuve du théorème 4.1

Théorème 4.1. *Le produit $\otimes$ des automates d'interface est une opération commututative et associative dans le cas où ils sont composables mutuellement.*

Démonstration. La preuve de la commututativité est triviale. Il reste à prouver l'associativité. Il est déjà prouvé que l'opérateur $\otimes$, défini dans le chapitre 3, est associatif. La seule différence entre la définition précédente et celle donnée dans le chapitre 3 est que les transitions synchronisées dans le produit de deux automates d'interface sont celles étiquetées par des actions partagées externes compatibles sémantiquement.

Les ensembles *Partagées*(A_1, A_2), *Partagées*(A_1, A_3) et *Partagées*(A_2, A_3) sont disjoints mutuellement à cause de la composabilité mutuelle des trois automates. Il est évident que les ensembles des états, des actions d'entrée de sortie et internes, et la fonction sémantique Ψ des automates $(A_1 \otimes A_2) \otimes A_3$ et $A_1 \otimes (A_2 \otimes A_3)$ sont les mêmes. Il reste donc à prouver que $\delta_{(A_1 \otimes A_2) \otimes A_3}$ égal à $\delta_{A_1 \otimes (A_2 \otimes A_3)}$. Pour prouver $E = F$, il suffit de prouver que $E \subseteq F$ et $F \subseteq E$. Pour prouver que $\delta_{(A_1 \otimes A_2) \otimes A_3} \subseteq \delta_{A_1 \otimes (A_2 \otimes A_3)}$, il faut montrer que (1) pour toute $(s, a, s') \in \delta_{(A_1 \otimes A_2) \otimes A_3}$, $(s, a, s') \in \delta_{A_1 \otimes (A_2 \otimes A_3)}$. Pour toute transition $((s_1, s_2, s_3), a, (s'_1, s'_2, s'_3))$ de $(A_1 \otimes A_2) \otimes A_3$, si $a \in$ *Partagées*(A_i, A_j) et *SemComp*$_a(A_i, A_j)$ tel que $i, j \in \{1, 2, 3\}$ et $i \neq j$, alors $(s_i, a, s'_i) \in \delta_{A_i}$, $(s_j, a, s'_j) \in \delta_{A_j}$ et $s_k = s'_k$ pour $k \in \{1, 2, 3\} \setminus \{i, j\}$. Par conséquent, il existe une transition $((s_j, s_k), a, (s'_j, s_k))$ de $A_j \otimes A_k$ telle que $((s_i, s_j, s_k), a, (s'_i, s'_j, s'_k))$ est une transition de $A_i \otimes (A_j \otimes A_k)$. Nous avons donc montrer que $\delta_{(A_1 \otimes A_2) \otimes A_3} \subseteq \delta_{A_1 \otimes (A_2 \otimes A_3)}$. L'autre sens de l'inclusion peut être vérifié en suivant la même procédure. □

Annexe B

Traduction des spécifications *ISIA* vers TLA$^+$

B.1 Spécification TLA$^+$ de l'automate d'interface sémantique du composant **Client**

MODULE *Client*

EXTENDS *Naturals*, *Sequences*, *TLC*
CONSTANTS *SessionStatut*, *PanierStatut*
VARIABLES *sess*, *panier*

$vars \triangleq \langle sess, panier \rangle$

$Init \triangleq \land sess = \text{“inactive”}$
$\land panier = \text{“vide”}$

$login \triangleq \land sess = \text{“inactive”}$
$\land$ UNCHANGED $vars$

$connecte \triangleq \land sess = \text{“inactive”}$
$\land sess' = \text{“active”}$
$\land$ UNCHANGED $\langle panier \rangle$

$echec \triangleq \land sess = \text{“inactive”}$
$\land$ UNCHANGED $vars$

$ajouterAuPanier \triangleq \land sess = \text{“active”}$
$\land panier = \text{“vide”} \lor panier = \text{“nonvide”}$
$\land$ IF $panier = \text{“vide”}$ THEN
$\land panier' = \text{“nonvide”}$
ELSE
$\land$ UNCHANGED $\langle panier \rangle$
$\land$ UNCHANGED $\langle sess \rangle$

$validerLoc \triangleq \land sess = \text{“active”}$
$\land panier = \text{“nonvide”}$

$\land$ UNCHANGED $vars$

$locValide \triangleq \land sess = \text{“active”}$
$\land$ UNCHANGED $vars$

$deconnexion \triangleq \land sess = \text{“active”}$
$\land sess' = \text{“inactive”}$
$\land$ UNCHANGED $\langle panier \rangle$

$Next \triangleq \lor login$
$\lor connecte$
$\lor echec$
$\lor deconnexion$
$\lor ajouterAuPanier$
$\lor validerLoc$
$\lor locValide$

$Liveness \triangleq \mathrm{SF}_{vars}(deconnexion) \land \mathrm{SF}_{vars}(connecte) \land \mathrm{SF}_{vars}(echec) \land \mathrm{SF}_{vars}(locValide)$

$Spec \triangleq Init \land \Box[Next]_{vars} \land Liveness$

$Invariant \triangleq$ TRUE

$Live \triangleq sess = \text{“active”} \leadsto sess = \text{“inactive”}$

B.2 Spécification TLA$^+$ de l'automate d'interface sémantique du composant Services

MODULE *Services*

EXTENDS *Naturals*, *Sequences*, *TLC*
CONSTANTS *SessionStatut*, *PanierStatut*
VARIABLES *sess*, *panier*, *tents*

$vars \triangleq \langle sess, tents, panier \rangle$

$Init \triangleq \land sess = \text{“inactive”}$
$\land tents = 2$
$\land panier = \text{“vide”}$

$login \triangleq \land sess = \text{“inactive”}$
$\land$ IF $tents \geq 0$ THEN
$\land tents' = tents - 1$
ELSE
$\land$ UNCHANGED $\langle tents \rangle$
$\land$ UNCHANGED $\langle sess, panier \rangle$

$$
\begin{array}{lll}
\mathit{connecte} & \triangleq & \land\ \mathit{sess} = \text{“inactive”} \\
& & \land\ \mathit{tents} \geq 0 \land \mathit{tents} < 2 \\
& & \land\ \mathit{sess}' = \text{“active”} \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{panier},\ \mathit{tents} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{echec} & \triangleq & \land\ \mathit{sess} = \text{“inactive”} \\
& & \land\ \text{UNCHANGED}\ \mathit{vars}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{ajouterAuPanier} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \quad \land\ \mathit{panier} = \text{“vide”} \lor \mathit{panier} = \text{“nonvide”} \\
& & \quad \land\ \text{IF}\ \mathit{panier} = \text{“vide”}\ \text{THEN} \\
& & \qquad \land\ \mathit{panier}' = \text{“nonvide”} \\
& & \quad \text{ELSE} \\
& & \qquad \land\ \text{UNCHANGED}\ \langle \mathit{panier} \rangle \\
& & \quad \land\ \text{UNCHANGED}\ \langle \mathit{sess},\ \mathit{tents} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{validerLoc} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“nonvide”} \\
& & \land\ \text{UNCHANGED}\ \mathit{vars}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{locValide} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“vide”} \\
& & \land\ \text{UNCHANGED}\ \mathit{vars}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{locNonValide} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“nonvide”} \\
& & \land\ \text{UNCHANGED}\ \mathit{vars}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{debiter} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“nonvide”} \\
& & \land\ \text{UNCHANGED}\ \mathit{vars}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{crdInsuff} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“nonvide”} \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{sess},\ \mathit{panier},\ \mathit{tents} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{nvCredit} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“nonvide”} \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{sess},\ \mathit{panier},\ \mathit{tents} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{dureeLocation} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{panier} = \text{“nonvide”} \\
& & \land\ \mathit{panier}' = \text{“vide”} \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{sess},\ \mathit{tents} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{deconnexion} & \triangleq & \land\ \mathit{sess} = \text{“active”} \\
& & \land\ \mathit{sess}' = \text{“inactive”} \\
& & \land\ \mathit{tents}' = 2 \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{panier} \rangle
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Next} & \triangleq \ \lor\ \mathit{login} \\
& \lor\ \mathit{connecte} \\
& \lor\ \mathit{echec} \\
& \lor\ \mathit{deconnexion} \\
& \lor\ \mathit{ajouterAuPanier} \\
& \lor\ \mathit{validerLoc} \\
& \lor\ \mathit{locValide} \\
& \lor\ \mathit{locNonValide} \\
& \lor\ \mathit{debiter} \\
& \lor\ \mathit{crdInsuff} \\
& \lor\ \mathit{nvCredit} \\
& \lor\ \mathit{dureeLocation}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Liveness} \triangleq \mathrm{SF}_{\mathit{vars}}(\mathit{deconnexion}) \land \mathrm{SF}_{\mathit{vars}}(\mathit{connecte}) \land \\
\qquad \mathrm{SF}_{\mathit{vars}}(\mathit{echec}) \land \mathrm{SF}_{\mathit{vars}}(\mathit{locValide}) \land \\
\qquad \mathrm{SF}_{\mathit{vars}}(\mathit{locNonValide}) \land \mathrm{SF}_{\mathit{vars}}(\mathit{crdInsuff}) \land \\
\qquad \mathrm{SF}_{\mathit{vars}}(\mathit{nvCredit}) \land \mathrm{SF}_{\mathit{vars}}(\mathit{debiter})
\end{array}
$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \quad \land \mathit{Liveness}$$

$$\mathit{Invariant} \triangleq \neg(\mathit{tents} = 2 \land \mathit{sess} = \text{“active”})$$

$$\mathit{Live} \triangleq \mathit{sess} = \text{“active”} \leadsto \mathit{sess} = \text{“inactive”}$$

B.3 Spécification TLA$^+$ de l'automate d'interface sémantique du composant **Services** raffiné

MODULE *ServiceRaf*

EXTENDS *Naturals*, *Sequences*, *TLC*
CONSTANTS *SessionStatut*, *PanierStatut*
VARIABLES *sess*, *asess*, *panier*, *tents*

$$\mathit{vars} \triangleq \langle \mathit{asess}, \mathit{sess}, \mathit{tents}, \mathit{panier} \rangle$$

$$
\begin{array}{ll}
\mathit{Init} & \triangleq \ \land\ \mathit{sess} = \text{“inactive”} \\
& \land\ \mathit{asess} = \text{“inactive”} \\
& \land\ \mathit{tents} = 2 \\
& \land\ \mathit{panier} = \text{“vide”}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{login} & \triangleq \ \land\ \mathit{sess} = \text{“inactive”} \\
& \land\ \mathit{asess} = \text{“inactive”} \\
& \land\ \text{IF } \mathit{tents} \geq 0 \text{ THEN} \\
& \qquad \land\ \mathit{tents}' = \mathit{tents} - 1 \\
& \quad \text{ELSE} \\
& \qquad \land\ \text{UNCHANGED } \langle \mathit{tents} \rangle \\
& \land\ \text{UNCHANGED } \langle \mathit{sess}, \mathit{asess}, \mathit{panier} \rangle
\end{array}
$$

$$\mathit{su} \triangleq \ \land\ \mathit{sess} = \text{“inactive”}$$

$\wedge\ asess = \text{“inactive”}$
$\wedge$ UNCHANGED $\langle sess,\ asess,\ panier,\ tents\rangle$

$connecte \triangleq$ $\wedge\ sess = \text{“inactive”}$
$\wedge\ asess = \text{“inactive”}$
$\wedge\ \vee\ \wedge$ IF $tents < 2 \wedge tents \geq 0$ THEN
$\wedge\ sess' = \text{“active”}$
ELSE
$\wedge$ UNCHANGED $\langle sess\rangle$
$\wedge$ UNCHANGED $\langle asess\rangle$
$\vee$
$\wedge\ asess' = \text{“active”}$
$\wedge$ UNCHANGED $\langle sess\rangle$
$\wedge$ UNCHANGED $\langle panier,\ tents\rangle$

$echec \triangleq$ $\wedge\ sess = \text{“inactive”}$
$\wedge\ asess = \text{“inactive”}$
$\wedge$ UNCHANGED $\langle sess,\ asess,\ panier,\ tents\rangle$

$ajouterAuPanier \triangleq$ $\wedge\ asess = \text{“inactive”}$
$\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“vide”} \vee panier = \text{“nonvide”}$
$\wedge$ IF $panier = \text{“vide”}$ THEN
$\wedge\ panier' = \text{“nonvide”}$
ELSE
$\wedge$ UNCHANGED $\langle panier\rangle$
$\wedge$ UNCHANGED $\langle asess,\ sess,\ tents\rangle$

$validerLoc \triangleq$ $\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“nonvide”}$
$\wedge$ UNCHANGED $\langle asess,\ sess,\ tents,\ panier\rangle$

$locValide \triangleq$ $\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“vide”}$
$\wedge$ UNCHANGED $\langle asess,\ sess,\ tents,\ panier\rangle$

$locNonValide \triangleq$ $\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“nonvide”}$
$\wedge$ UNCHANGED $\langle asess,\ sess,\ panier,\ tents\rangle$

$debiter \triangleq$ $\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“nonvide”}$
$\wedge$ UNCHANGED $\langle asess,\ sess,\ panier,\ tents\rangle$

$crdInsuff \triangleq$ $\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“nonvide”}$
$\wedge$ UNCHANGED $\langle asess,\ sess,\ panier,\ tents\rangle$

$nvCredit \triangleq$ $\wedge\ sess = \text{“active”}$
$\wedge\ panier = \text{“nonvide”}$

$$\land \text{ UNCHANGED } \langle asess,\ sess,\ panier,\ tents \rangle$$

$$\begin{aligned} dureeLocation \triangleq\ & \land sess = \text{“active”} \\ & \land panier = \text{“nonvide”} \\ & \land panier' = \text{“vide”} \\ & \land \text{UNCHANGED } \langle asess,\ sess,\ tents \rangle \end{aligned}$$

$$\begin{aligned} validerInscription \triangleq\ & \land asess = \text{“active”} \\ & \land \text{UNCHANGED } vars \end{aligned}$$

$$\begin{aligned} deconnexion \triangleq\ & \land sess = \text{“active”} \lor asess = \text{“active”} \\ & \land \text{IF } sess = \text{“active”} \text{ THEN} \\ & \quad \land sess' = \text{“inactive”} \\ & \quad \land tents' = 2 \\ & \quad \land \text{UNCHANGED } \langle asess \rangle \\ & \text{ELSE} \\ & \quad \land asess' = \text{“inactive”} \\ & \quad \land \text{UNCHANGED } \langle sess,\ tents \rangle \\ & \land \text{UNCHANGED } \langle panier \rangle \end{aligned}$$

$$\begin{aligned} Next \triangleq\ & \lor login \lor su \lor connecte \\ & \lor echec \lor deconnexion \\ & \lor ajouterAuPanier \\ & \lor validerLoc \lor locValide \lor locNonValide \\ & \lor debiter \lor crdInsuff \lor nvCredit \\ & \lor dureeLocation \\ & \lor validerInscription \end{aligned}$$

$$\begin{aligned} Liveness \triangleq\ & \mathrm{SF}_{vars}(deconnexion) \land \mathrm{SF}_{vars}(connecte) \land \\ & \mathrm{SF}_{vars}(echec) \land \mathrm{SF}_{vars}(locValide) \land \\ & \mathrm{SF}_{vars}(locNonValide) \land \mathrm{SF}_{vars}(crdInsuff) \land \\ & \mathrm{SF}_{vars}(nvCredit) \land \mathrm{SF}_{vars}(debiter) \land \mathrm{SF}_{vars}(su) \end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \quad \land Liveness$$

$$Invariant\_1 \triangleq \neg(tents = 2 \land sess = \text{“active”})$$
$$Invariant\_2 \triangleq \neg(sess = \text{“active”} \land asess = \text{“active”})$$

$$Live\_1 \triangleq sess = \text{“active”} \leadsto sess = \text{“inactive”}$$
$$Live\_2 \triangleq asess = \text{“active”} \leadsto asess = \text{“inactive”}$$
