Contribution à la spécification et à la vérification des logiciels à base de composants : enrichissement du langage de données de Kmelia et vérication de contrats

El-Habib Mohamed Messabihi

Université de Nantes

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur de l'Université de Nantes

Spécialité : **Informatique**

préparée au laboratoire **d'Informatique de Nantes Atlantique**

dans le cadre de l'École Doctorale **Sciences et Technologies de l'Information et des Matériaux**

présentée et soutenue publiquement
par

Mohamed El-Habib Messabihi

le 04 Juillet 2011

Titre :

Contribution à la spécification et à la vérification des logiciels à base de composants : enrichissement du langage de données de Kmelia et vérification de contrats

Directeur de thèse : **Christian Attiogbé**
Co-encadrant de thèse : **Pascal André**

Jury

Président :	...	...
Rapporteurs :	**Antoine Beugnard**	Professeur à Telecom Bretagne
	Jean-Paul Bodeveix	Professeur à l'Université Paul Sabatier
Examinateurs :	**Stephane Ducasse**	Directeur de Recherches INRIA
	Olga Kouchnarenko	Professeur à LIFC-UFR-ST
Directeur de thèse :	**Christian Attiogbé**	Professeur à l'Université de Nantes
Co-encadrant :	**Pascal André**	Maître de conférences à l'Université de Nantes

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Cette thèse a été effectuée au sein de l'équipe AeLoS [1] dont les travaux de recherche se focalisent sur l'élaboration des concepts, des méthodes et des techniques outillées afin d'assurer la correction des systèmes complexes. L'équipe AeLoS a développé un *framework* basé sur un modèle de composants appelé Kmelia ainsi qu'une boîte à outils COSTO [2] offrant des possibilités de vérification de propriétés. Les objectifs de cette thèse sont d'enrichir le *framework* Kmelia/COSTO afin de répondre aux besoins suivants :

- *expressivité :* la capacité de couvrir différents aspects (structurel, fonctionnel et dynamique) dans les spécifications des systèmes à composants,
- *correction :* la possibilité de vérifier formellement (tôt dans le cycle de développement) des propriétés prescrites liées aux différents aspects retenus,
- *outillage :* la possibilité d'assister les utilisateurs du *framework* dans les différentes tâches de spécification et de vérification.

Nous détaillerons ces objectifs après avoir présenté le contexte général de la thèse.

1.1 Contexte et problématique

L'informatique joue un rôle primordial dans les différents secteurs d'activité (télécommunications, transports, énergies, transactions financières, santé, *etc.*). L'omniprésence des systèmes informatiques dans la vie quotidienne (automobile, téléphonie, moyens de paiement, GPS, *etc.*) rend la société moderne de plus en plus assujettie à ces systèmes. Des dysfonctionnements dans ces derniers peuvent avoir des répercussions bien plus graves qu'un simple désagrément pour l'utilisateur (*e.g.* la réinitialisation d'un GPS), allant dans certains cas jusqu'à des accidents graves avec des dégâts catastrophiques en terme d'économie ou de vie humaine [3] (*e.g.* le *crash* d'un avion). La *vérification* du bon fonctionnement de ces systèmes devient donc un défi de plus en plus difficile à relever face à leur complexité croissante. Le

1. Architecture et Logiciels Sûrs. Anciennement appelée COLOSS (COmposant et LOgiciels Sûrs).

2. COmponent Study TOolbox.

3. Parmi les *bugs* les plus célèbres, on pourra citer celui de la fusée Ariane-5 (700 millions de dollars de perdus) ou encore celui du missile Patriot (28 morts) causés tous deux par des erreurs de conversion de représentation des nombres par les systèmes [JM97].

développement de ces derniers nécessite ainsi des efforts de conception et de vérification conséquents et complexes, qui vont à l'encontre des objectifs industriels de réduction de coûts et de délais de production.

Pour pallier ce problème, la communauté du génie logiciel cherche en permanence à élaborer des méthodes, des techniques et des outils permettant de réduire le coût et le délai de développement de tels systèmes tout en garantissant les qualités souhaitées par leurs utilisateurs [Jau94]. Ainsi, face à cette complexité croissante, deux principes fondamentaux ont été clairement identifiés, à savoir : la *modularisation* et la *réutilisation* [GJM02]. Le premier (*modularisation*) vise à encapsuler et à isoler, le plus possible, chaque fonctionnalité dans des briques logicielles différentes. Le logiciel résultant est donc vu comme un assemblage de ces briques qui fournissent dans leur ensemble toutes les fonctionnalités attendues par les utilisateurs. Le second (*réutilisation*) consiste à autoriser l'utilisation de ces mêmes briques logicielles afin de concevoir d'autres logiciels plus complexes. Ce principe permet ainsi d'accélérer le développement du logiciel.

Quant aux exigences croissantes en matière de fiabilité de tels systèmes, il est nécessaire de disposer, dans le cycle de développement, d'une démarche sûre pour s'assurer de la *correction* des systèmes. Cette démarche permettra de découvrir systématiquement les comportements indésirables qui doivent être corrigés. Pour y parvenir, deux approches sont généralement pratiquées. La première est le test qui, bien qu'utile, ne permet pas de couvrir toutes les possibilités qui peuvent se produire lors de l'exécution du système dans la réalité. Le nombre de cas possibles est habituellement si grand que nous ne pouvons en tester qu'une infime proportion. De plus, l'absence de violations de propriétés prévues par le test ne signifie pas que le comportement du système est correct. La seconde approche est la vérification formelle qui est, de fait, reconnue pour remédier à ce problème en procurant la capacité d'analyser exhaustivement les systèmes selon des caractéristiques prescrites exprimant leur comportement désiré, mais au prix d'une complexité et d'un coût importants. Cependant, la vérification formelle est d'autant plus efficace et rentable, qu'elle est outillée et exploitée au plus tôt dans le cycle de développement. La figure 1.1 situe la problématique de la thèse par rapport aux domaines du CBSE, de la conception par contrat et des méthodes formelles.

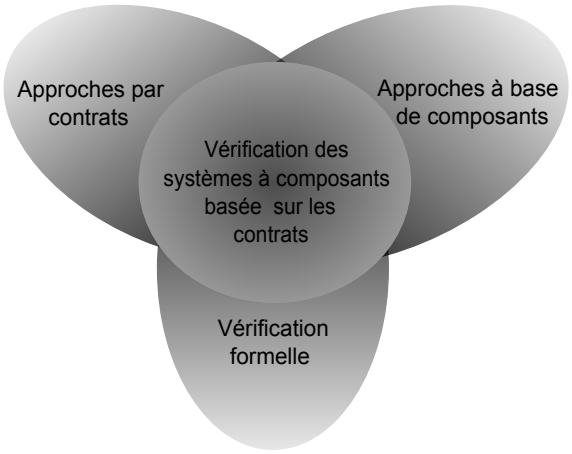


FIGURE 1.1 – Contexte général de notre approche

Les approches à base de composants

La mise en œuvre des deux principes susmentionnés (*modularisation* et *réutilisation*) dans les approches de développement logiciel a fait émerger un nouveau paradigme appelé CBSE *(Component Based Software Engineering)*. Dans le domaine des composants matériels (électroniques, mécaniques, *etc.*), cela fait maintenant plusieurs années que ces méthodes sont bien assimilées et largement employées avec succès. Dans le domaine du logiciel, les résultats sont encore peu concluants [CL00]. Cependant, les approches à base de composants permettent d'ores et déjà une meilleure structuration de l'application et fournissent des moyens accrus de modularité et de réutilisation. Chaque composant explicite les fonctionnalités qu'il offre et qu'il requiert à travers ses interfaces. Les composants peuvent ainsi être stockés dans des bibliothèques (COTS [1]) pour qu'ils puissent être réutilisés et connectés les uns aux autres, assurant ainsi un gain en temps et en coût lors du développement.

Dans ce paradigme, les composants sont souvent perçus comme des boîtes noires et ne sont accessibles qu'à travers leurs points d'accès (souvent appelés interfaces), il est donc nécessaire d'enrichir lesdites interfaces afin de permettre la réutilisation des composants et leur vérification. À l'instar de Szyperski [Szy02] qui a défini un composant comme : « *A software component is a unit of composition with* ***contractually*** *specified* ***interfaces*** *and explicit context dependencies only* », nous avons choisi l'approche basée sur les contrats pour répondre à cette problématique.

L'approche par contrat (appliquée aux composants)

La conception par contrats (DbC pour *Design by Contract*) [Mey92] est une méthode de conception logicielle. Bien qu'elle soit bien formalisée dans certaines approches, son utilisation dans les composants n'a toujours pas atteint sa maturité.

En génie logiciel, le terme *contrat* est souvent confondu avec les assertions (pré, post-conditions ou invariants), mais la définition des contrats va bien au-delà du mécanisme des assertions qui n'a que peu en commun avec les aspects méthodologiques qu'offrent les contrats au sens large. Ainsi, appeler correctement (avec la bonne syntaxe et les bons types de données échangées) un service (ou une opération) est aussi une forme de contrat pour valider le service. De plus, même si les assertions permettent d'améliorer le niveau de confiance des systèmes, cela n'est garanti que sous une hypothèse forte assurant que l'exécution des services (ou opérations) est atomique, ou du moins séquentielle. Néanmoins, les systèmes sont de plus en plus complexes et intrinsèquement concurrents. Des modèles exprimant des contraintes de communication et de synchronisation sont donc nécessaires pour assurer le bon fonctionnement de tels systèmes. Ces contraintes constituent une forme de contrat (tel service doit envoyer -ou recevoir- telle donnée à tel moment). D'autres formes de contrats dits de QoS (*Quality of Service*) peuvent également être considérées [BJPW99b] ; mais les propriétés exprimées par ces derniers sont, la plupart du temps, fortement liées à des informations qui ne sont connues qu'au moment de l'exécution (*runtime*) ou qui dépendent de la plate-forme utilisée. Pour ces raisons,

1. *Component-Off-The-Shelf*

les contrats de QoS ne seront pas traités dans cette thèse.

Les contrats permettent l'expression explicite d'un certain nombre d'obligations relatives au bon fonctionnement des différents services des composants. L'approche par contrat que nous préconisons dans cette thèse est basée non pas uniquement sur un ensemble de spécifications mais également sur des mécanismes de vérification outillés afin de garantir certaines propriétés aux systèmes modélisés. Par conséquent, elle est en étroite liaison avec les méthodes formelles.

La vérification formelle

Les méthodes formelles [Win90] sont définies comme un ensemble de notations pourvues d'une sémantique formelle (basée sur des modèles mathématiques) et d'outils, permettant de spécifier sans ambiguïté un système complexe dans les différentes étapes de son cycle de développement (spécification de besoin, conception architecturale, conception détaillée, *etc.*), et de valider (simuler et vérifier formellement) un certain nombre de propriétés. De plus, quand elles sont utilisées tôt dans le cycle de développement, elles permettent de découvrir des failles de conception qui, autrement, ne pourraient être découvertes qu'avec de coûteuses étapes de test et de débogage. Nous distinguons deux familles de méthodes de vérification formelle : les méthodes basées sur l'évaluation des modèles (*model checking*) et les méthodes basées sur la preuve de théorème (*theorem proving*).

Vérification par *model checking*

L'évaluation de modèle, ou le *model checking* [CG87], consiste à construire un modèle fini d'un système et à vérifier de façon automatique qu'une propriété donnée est vraie ou fausse dans ce modèle. Le *model checker* procède par exploration des états. La plupart des *model-checkers* sont capables de générer une réponse positive si la propriété est garantie pour tous les comportements du modèle, ou une réponse négative, complétée par un contre-exemple illustratif, en cas de violation de la propriété. Bien que cette approche ait l'avantage d'être exhaustive, elle est limitée par le problème classique de l'explosion combinatoire du nombre d'états.

Vérification par preuve

La preuve de théorèmes, ou *theorem proving* [Hoa69], est une technique consistant à exprimer le système et les propriétés recherchées sous forme de formules dans une logique mathématique, puis procéder à une recherche de la preuve de ces propriétés. Elle établit, par une suite finie d'étapes de raisonnement appelées inférences, qu'une certaine propriété est la conséquence logique d'axiomes, de règles de déduction et d'un ensemble d'hypothèses décrivant le système étudié. La preuve peut être appliquée à toutes les phases de développement du système. Il est possible, par exemple, de chercher à prouver qu'une spécification satisfait certaines propriétés, qu'un programme est correct par rapport à une spécification détaillée (preuve de programme), ou qu'il se termine toujours sous certaines conditions (preuve de terminaison).

Que l'on utilise la première approche ou la seconde, les propriétés dont on désire vérifier la validité doivent être exprimées par le spécifieur du système. Il est donc nécessaire de fournir à ce dernier un langage précis et non ambigu pour l'expression de ces propriétés.

1.2 Objectifs et contributions

Comme nous l'avons déjà évoqué précédemment, les objectifs de cette thèse sont d'enrichir le *framework* Kmelia/COSTO afin de pouvoir répondre aux questions suivantes :

- comment vérifier la *conformité d'un service* ? c'est-à-dire vérifier que le comportement de chaque service est conforme à sa spécification (ce que l'on attend de lui) ;
- comment vérifier la *cohérence d'un composant* ? c'est-à-dire vérifier, d'une part, que le composant préserve un certain nombre de propriétés quelque soit son implantation et, d'autre part, que son protocole (la manière dont il doit être utilisé) est cohérente ;
- comment vérifier la *correction* d'un assemblage de plusieurs composants ? c'est-à-dire en vérifiant la compatibilité entre les composants en terme de respect des engagements et des garanties des uns et des autres ;
- comment vérifier la sûreté de la *promotion* d'un service lorsqu'il est adapté à l'environnement du composite ?

Afin d'y parvenir, notre proposition repose sur la mise en œuvre conjointe des approches à base de composants et à base de contrats. Il est donc indispensable de comprendre les concepts de ces deux approches. La première contribution de cette thèse est la réalisation d'une étude détaillée sous forme de synthèse des principales approches à base de composants et des approches par contrats en comparant les concepts présents dans un certain nombre de langages, de modèles et de plates-formes. L'intérêt de cette synthèse est non seulement de mettre en évidence les concepts-clés (souvent confondus) dans les deux approches, mais également d'identifier les manques dans les propositions existantes.

Outre cette synthèse, la présente thèse regroupe des contributions théoriques et pratiques basées sur les travaux menés conjointement autour du modèle Kmelia et de sa boîte à outils COSTO. Kmelia est un modèle et un langage à composants multi-services. Il partage des caractéristiques communes avec les approches à composants [AG97, MT00] : les composants, les services, les assemblages et les composites. Il a l'avantage d'être à la fois abstrait (indépendant d'un environnement d'exécution) et formel. Il peut donc servir de support pour l'étude de propriétés de modèles de composants et de services. En outre, il est doté d'une boîte à outils offrant de nombreuses vérifications, notamment celle de la compatibilité dynamique. Cependant, l'élaboration du modèle Kmelia suit une approche incrémentale à partir d'un noyau. À notre arrivée, il ne disposait pas de définition formelle de la partie fonctionnelle (langage de données et vérification associée) qui est fondamentale pour l'expression des contrats dans le paradigme composant. Nous avons abordé cette problématique dans les trois contributions suivantes (cf. figure 1.2).

1. **Enrichissement du modèle Kmelia par un langage de données.** Nous avons proposé un langage qui couvre la définition des types de données, des expressions, des assertions (pré/post-conditions, invariants, *etc.*), des communications, du typage des composants et des assemblages. Les propriétés d'intérêt sont la sûreté de fonctionnement des services et la préservation des propriétés des services dans les assemblages. Le modèle de contrat que nous avons introduit peut s'adapter à d'autres modèles de composants. Ce premier axe a abouti aux résultats publiés dans [AAM09, AAA+09].
2. **Définition et automatisation de techniques de vérification basées sur la notion de contrat multi-niveaux.** Nous avons proposé des techniques de vérification pour prouver la préservation de l'intégrité des composants par les services, la sûreté de fonctionnement des services eux-mêmes et le respect des contrats d'assemblage [MAA10c] et de promotion [AAM10]. Ces techniques sont basées sur la notion de contrats multi-niveaux que nous avons proposée pour, d'une part, expliciter la notion de responsabilité des différents éléments de modélisation (service, composant et assemblage, *etc.*) et, d'autre part, maîtriser la complexité de la spécification et de la vérification [MAA10a]. Le processus de vérification est mis en œuvre en utilisant les outils de preuve associés à la méthode B [MAA10b].
3. **Contribution au développement de la boîte à outils COSTO.** Nous avons étendu la grammaire du modèle Kmelia pour intégrer le langage de données que nous avons introduit. Nous avons également développé un nouveau *plug-in* appelé Kml2B assurant une passerelle entre l'outil COSTO et les différents outils de vérification associés à la méthode B. Ces extensions ont été, en partie, présentées dans [AAM11, AMAA11].

1.3 Organisation du manuscrit

La suite de ce manuscrit est organisée comme suit :

Chapitre 2. Ce chapitre expose les principaux concepts sous-jacents aux modèles de composants. Il donne également un aperçu des approches basées sur les contrats. Ensuite, il passe en revue de quelques modèles de composants pour étudier la manière dont ils autorisent l'expression des contrats et la vérification des propriétés. Enfin, il dresse un bilan sur l'état de l'art.

Chapitre 3. Ce chapitre introduit le modèle à composants Kmelia. D'abord, une présentation informelle du modèle est donnée ; ensuite les concepts de base sont présentés progressivement. Enfin, une synthèse des propriétés vérifiées ainsi que la boîte à outils COSTO sont présentées. Le tout est illustré à travers un cas d'étude inspiré du CoCoME (Common Component Modelling Example). Le modèle Kmelia sera utilisé pour répondre aux problématiques telles que l'intégration du langage de données (cf. chapitre 4), la vérification des composants et leur assemblage (cf. chapitre 5 et chapitre 6).

Chapitre 4. Ce chapitre présente un langage de données permettant de décrire l'aspect fonctionnel de Kmelia qui couvre la définition des types de données,

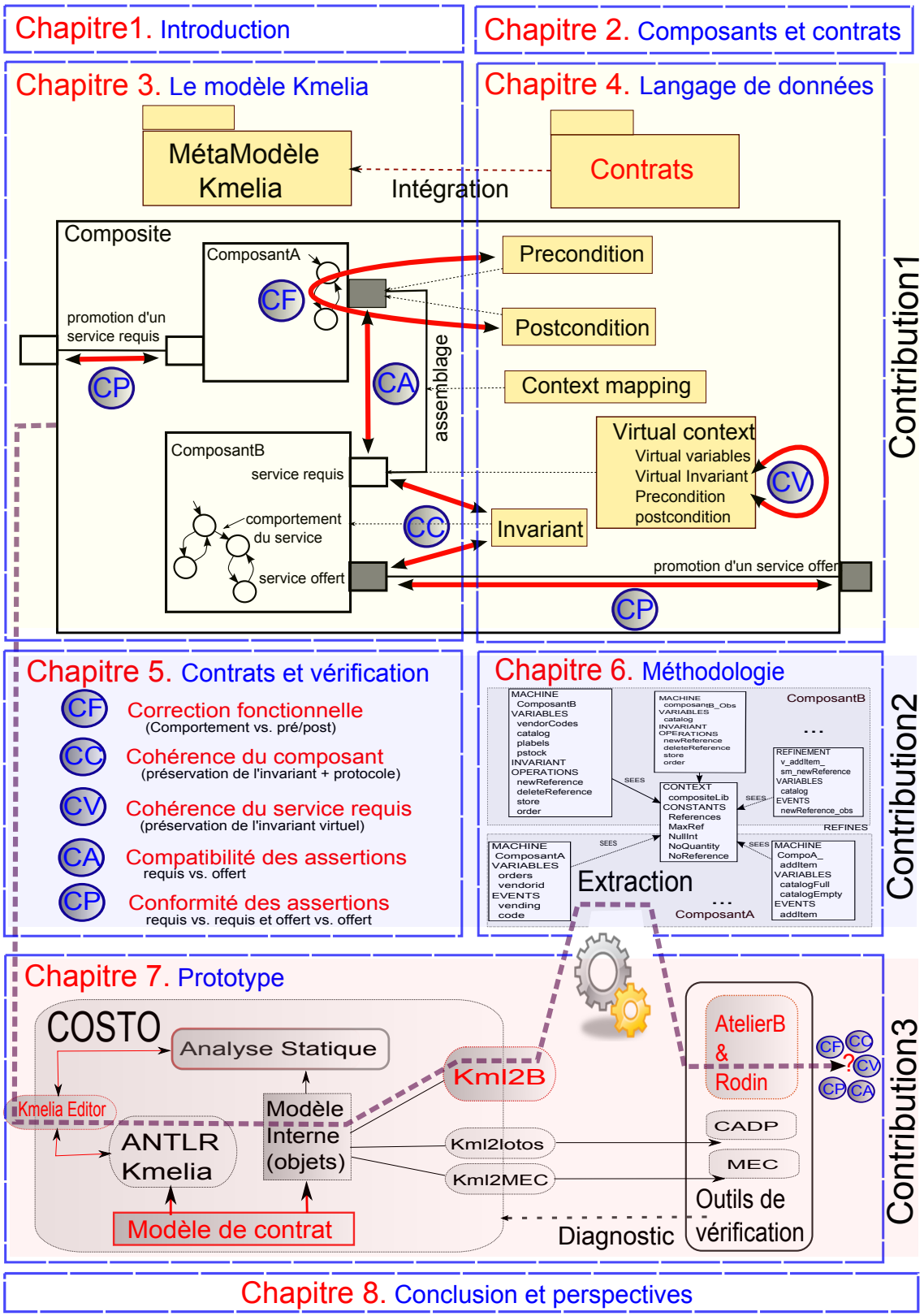


FIGURE 1.2 – Représentation schématique des contributions et de la structure de cette thèse.

l'état des composants, les expressions, les assertions des services (pré/post-conditions), les communications, le typage des composants et des assemblages. Il met en évidence l'impact de l'aspect fonctionnel sur le processus de vérification des composants et de leur composition. Une étude de cas est présentée progressivement pour illustrer les différentes propositions.

Chapitre 5. Ce chapitre montre comment tirer parti des contrats dans la vérification des systèmes à composants. Dans un premier temps, il introduit la notion de contrats multi-niveaux et ensuite, il présente un processus de développement des composants basée sur l'utilisation des contrats. Enfin, il traite deux niveaux de vérification : d'une part, les vérifications intra-composant (composants individuels et leurs services) et, d'autre part, les vérifications inter-composants (liées essentiellement aux assemblages et aux composites).

Chapitre 6. Ce chapitre présente une automatisation de nos techniques de vérification. Il donne un bref aperçu de la méthode B et de ses outils qui ont servi de base pour le processus de vérification. Il présente, ensuite, les règles de traduction des composants et services Kmelia en des spécifications B et exhibe les obligations de preuve pour chaque propriété. Enfin, il montre comment les propriétés, exprimées par contrats d'assemblage et de promotion, sont vérifiées en utilisant la méthode B.

Chapitre 7. Ce chapitre décrit le prototype que nous avons développé durant cette thèse, ainsi que les expérimentations permettant de valider notre approche. Le prototype permet de concrétiser l'ensemble de nos propositions autour de Kmelia. Il montre comment sont implantées les différentes propositions présentées dans les chapitres précédents, tant pour l'intégration du langage de données et de contrats que pour l'extraction des spécifications B. L'ensemble de ces implantations a été intégré à la plate-forme COSTO sous forme de *plugin*. Le tout est illustré par une application de gestion de stock.

Chapitre 2

Composants et contrats logiciels

Nous nous intéressons, dans le cadre de cette thèse, au problème de la vérification des logiciels à base de composants. D'abord, nous avons étudié les modèles de composants existants afin d'identifier les caractéristiques et les concepts de base ainsi que les propriétés spécifiques aux systèmes à base de composants.

Avec cet objectif en tête, nous avons étudié différents modèles (et plates-formes) à base de composants et différentes techniques de vérification. Comme beaucoup d'auteurs de la communauté, nous sommes convaincus que pour vérifier les systèmes à base de composants, il faut travailler sur des modèles suffisamment abstraits (indépendamment de toute implantation).

Lors de cette étude, nous avons pu identifier un manque considérable au niveau de la vérification des systèmes à base de composants de façon générale, et en particulier au niveau de la vérification tôt dans le cycle de développement de tels systèmes. Nous avons alors étudié différentes solutions existantes qui pourraient être appliquées aux composants pour combler ce manque. Les techniques de vérification, basées sur les contrats, ont particulièrement attiré notre attention, puisqu'elles nous semblent être parfaitement compatibles avec notre objectif.

Ainsi nous allons, dans un premier temps, étudier les principaux concepts sous-jacents aux modèles de composants (cf. sections 2.2). Ensuite, nous envisagerons l'approche basée sur les contrats (cf. section 2.3). Nous ferons par la suite une revue de différents modèles de composants pour étudier la manière dont ils autorisent l'expression et la vérification des propriétés (cf. section 2.4). Enfin, nous présenterons un bilan sur l'état de l'art (cf. section 2.5) et conclurons le chapitre.

2.1 Avènement des composants logiciels

La présence du terme *composant* dans le vocabulaire informatique remonte à la fin des années 60. Il a cependant d'abord désigné des fragments de code alors qu'aujourd'hui, il englobe toute une unité de réutilisation. À l'origine, l'approche à base de composants est fortement inspirée des composants de circuits électroniques. L'idée de circuits logiciels intégrés conduit à la notion d'éléments logiciels qui peuvent être branchés ou débranchés d'un circuit plus complexe, remplacés et/ou configurés, *etc.*

La figure 2.1 résume l'évolution des concepts manipulés par la communauté in-

formatique au cours des cinquante (ou plus) dernières années. Ces concepts sont de plus en plus abstraits et donc simples à utiliser par les concepteurs d'applications. En contrepartie les mécanismes sous-jacents sont de plus en plus complexes à mettre en œuvre par les concepteurs des modèles les supportant.

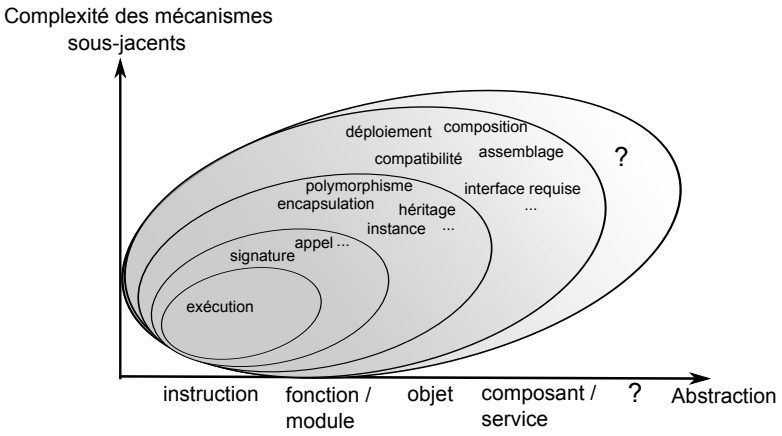


FIGURE 2.1 – Comparaison des concepts.

L'instruction est le concept le plus simple et le plus proche de la machine. Mais il est très difficile d'appréhender un programme d'un millier de lignes, et la vérification de propriétés de sûreté est assez complexe.

La procédure (ou fonction) permet de s'éloigner un peu du cœur de la machine : les instructions y sont regroupées selon une fonctionnalité précise. La procédure peut être réutilisée à d'autres endroits du programme, voire par d'autres programmes. Toutefois, les procédures manipulent des données qui sont décrites à part ; les programmes sont plutôt conçus en termes d'algorithmes que de données manipulées.

Les objets, dans les années 80, ont recentré les concepts sur ces données : un objet est une représentation des données, avec des méthodes permettant de les manipuler. La conception des applications s'en est trouvée améliorée : un logiciel est décrit en termes de relations entre objets permettant ainsi un passage d'une simple décomposition fonctionnelle du système à une véritable collaboration d'objets. Le concept d'objet a donc permis une abstraction supplémentaire et une modularité accrue, l'inconvénient est qu'il est plus complexe à mettre en œuvre que les concepts précédents.

Même si l'avènement des objets a permis de faire un bond en avant assez conséquent dans la conception des logiciels, ce concept a rapidement montré ses limites. En effet, l'approche orienté-objet souffre d'un manque de mécanismes et d'outils permettant la composition d'objets [Mey97]. Cette dernière est nécessaire lorsque des considérations de passage à l'échelle doivent être prises en compte. Cette limitation est accentuée par le couplage implicite qui existe entre les objets. Ce couplage est introduit par l'existence, au sein du code même des objets, de références vers d'autres objets. Il est ainsi très difficile d'identifier, au sein d'une application mettant en jeu un grand nombre d'objets, les dépendances (les requis) d'un objet vis-à-vis

des autres. Ces dépendances sont enfouies dans le code source, rendant difficile la possibilité d'identifier des objets précis en vue de leur réutilisation dans une autre application. Le paradigme objet ne fournit donc pas de réponses satisfaisantes au principe de réutilisation et de substitution. C'est pourquoi la conception dans le but de réutiliser demande des mécanismes supplémentaires à ceux nécessaires à une conception ayant pour unique but d'utiliser.

L'ingénierie logicielle à base de composants CBSE (*Component Based Software Engineering*) apporte des éléments de réponses à cette perspective. En effet, depuis la première conférence sur l'ingénierie du logiciel organisée par l'OTAN en 1968, les efforts pour définir clairement ce qu'est un composant logiciel ont été nombreux (notamment dans [Fro02] et [Tho06]).

Ces dernières années, un pas supplémentaire a été franchi grâce aux composants. En effet, les composants peuvent regrouper des objets dans des morceaux de code (ou d'application), que l'on rassemble (connecte) pour construire le logiciel final [HC01]. L'abstraction résultante est nettement plus élevée : l'application est vue comme des pièces d'un *puzzle* que l'on assemble. La réutilisation est véritablement pratiquée : une application paramètre le composant pour s'en servir conformément à ses besoins spécifiques. En revanche, la complexité sous-jacente est importante : apparaissent les notions de compatibilité des interfaces de composants, de composition de composants et de leurs services, de gestion de leurs types et de leurs instances, de leur déploiement, *etc.* Même s'il était possible d'appréhender l'appel de méthode d'un objet dans ses moindres détails, il est plus difficile de connaître toutes les fonctionnalités engendrées lors de l'utilisation d'un service d'un composant. Mais l'avantage qui en résulte est que l'application n'est plus conçue en termes de méthodes appliquées à un objet, mais réellement de services fournis par le composant, pour les besoins de l'application. L'interaction avec un composant n'est pas à considérer comme un simple appel de procédure ou de méthode, mais plutôt comme un véritable dialogue avec le composant. En résumé, un modèle de composant s'approche plus du modèle mental du développeur que les modèles précédents. Dans le meilleur des mondes, le développeur d'applications n'aurait plus qu'à chercher ses composants sur une étagère, configurer leurs paramètres pour les adapter à ses besoins, et les interconnecter pour obtenir son application finale.

Il est important de mentionner que les descriptions d'architecture basées sur les concepts présentés ci-dessus forment des couches conceptuelles successives. On peut concevoir un composant (par exemple, distributeur automatique de billets) reposant sur des services (par exemple, consultation, retrait, dépôt, *etc.*), eux-mêmes reposant sur des objets (par exemple, compte, client, *etc.*). Chaque catégorie d'architecture offre une couche d'abstraction sur la précédente permettant de masquer progressivement les détails pour se focaliser sur des objectifs proportionnels et sur des préoccupations relatives à la vue que l'on a sur le système.

2.2 Modèles de composants

Un modèle est une abstraction d'un système autorisant les prédictions ou les inférences [Küh06]. Grâce à son caractère abstrait, un modèle permet de faire face

à la complexité du système qu'il représente et permet ainsi d'en faciliter la compréhension et le raisonnement sur ses concepts sous-jacents. Pour réduire la complexité, un modèle se focalise sur un nombre réduit d'aspects du système (structure, comportement, interaction, *etc.*). Un ensemble de modèles permet alors de décrire un système sous différents angles.

Dans le contexte du CBSE, un modèle de composants définit la manière dont les composants doivent être construits et la manière dont ils doivent être assemblés les uns avec les autres. Le modèle de composants définit donc la structure, les interactions et les principes de composition que les composants doivent respecter pour être conformes au modèle [CH01].

D'après Lau et Wang [LW07], un modèle de composants logiciels est caractérisé par : une syntaxe (graphique ou textuelle) servant à décrire la structure du composant, indépendamment de la syntaxe d'implantation ; une sémantique décrivant les composants (définition des interfaces, services fournis/requis, opérations, *etc.*) et un principe de composition, détaillant la manière de composer les composants.

Actuellement, il n'existe pas de terminologie ni de critères standardisés qui permettent d'identifier clairement un composant. Il n'y a pas non plus un modèle unifié de composants, seulement des modèles de composants. Ainsi, un composant est identifiable comme tel, toujours par rapport à son modèle de composants au sens de [LW07].

Dans la pratique, la plupart des modèles se spécialisent pour un domaine particulier ou bien ne couvrent pas l'ensemble du cycle de vie du composant depuis sa spécification, considérée comme le plus haut niveau d'abstraction, jusqu'à son code binaire, représentant le plus bas niveau. Dans le domaine des architectures logicielles, on se focalise surtout sur la description haut niveau d'un système logiciel afin de pouvoir analyser ses propriétés. En d'autres termes, on s'intéresse surtout aux phases de conception et d'analyse du cycle de vie du logiciel. En revanche, dans le domaine des technologies de composants, on s'intéresse le plus souvent à l'aspect intergiciel (distribution, persistance, sécurité, déploiement, *etc.*). Ainsi, on se limite généralement à la phase d'implantation, de déploiement et d'exécution dans le cycle de vie. Naturellement, ces deux domaines se complètent. Il est donc tout à fait envisageable de commencer le développement d'un logiciel par la description de son architecture dans les phases d'analyse/conception pour l'implanter avec des composants logiciels exécutables.

✎ Note 1.

Dans cette thèse, nous nous intéressons plutôt aux modèles de composants abstraits pour pouvoir raisonner sur les propriétés des composants et leur assemblage. Nous ne nous intéressons pas aux aspects de déploiement ou d'exécution des composants.

2.2.1 Composant

Selon le *Petit Robert*, un composant est un *« élément qui entre dans la composition de quelque chose et qui remplit une fonction particulière »* .

L'utilisation de l'approche à base de composants dans le développement d'applications logicielles soulève rapidement des questions : quelle est la définition d'un composant ? Quelle est sa granularité, sa portée ? Comment distinguer et rechercher les composants de manière rigoureuse, les manipuler, les assembler, les vérifier et les tester ? Comment les réutiliser, les installer dans des contextes matériels et logiciels variant dans le temps, les administrer et les faire évoluer ? *etc.*

2.2.1.1 Définitions

De nombreuses définitions ont été proposées pour définir ce qu'est un composant. Nous retenons celle de *Szyperski* [Szy02], donnée initialement à l'occasion d'un atelier à ECOOP [1] en 1996. Cette définition nous semble être la plus consensuelle et la plus répandue dans la littérature. Ainsi un composant est défini comme suit :

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.

Cette définition met en avant trois propriétés fondamentales d'un composant logiciel. Premièrement, un composant est une entité qui décrit explicitement les fonctionnalités qu'il offre par le biais d'interfaces contractualisées (c'est-à-dire dotées de contrats), ainsi que ses dépendances (c'est-à-dire les fonctionnalités qu'il requiert). Deuxièmement, un composant est une entité composable. Concrètement, cela signifie qu'une application à base de composants est un assemblage de composants. De plus, cette composition doit satisfaire les exigences décrites par les interfaces contractualisées des composants. Troisièmement, un composant est une entité capable d'être déployée sur une plate-forme d'exécution, indépendamment des autres composants.

Ces aspects sont mis en avant notamment par Bertrand Meyer [Mey03] : *un composant est un élément logiciel (unité modulaire) qui satisfait les conditions suivantes : (i) il peut être utilisé par d'autres éléments logiciels, ses clients, (ii) il possède une description de son usage permettant aux concepteurs de clients de l'utiliser, (iii) il n'est lié à aucun ensemble fixe de clients.*

Dans les définitions précédentes, on se focalise sur les aspects composition, interfaces contractuelles, modularité, substitution, interfaces requises et fournies. En effet, si on se place dans un contexte d'assemblage, alors les composants sont supposés déjà développés et l'on voudrait les assembler pour former une application. Ces composants spécifient des interfaces sous forme de contrats, et peuvent être déployés séparément. On peut également se placer dans un contexte de modélisation objet où l'on perçoit le système comme un ensemble d'unités modulaires. Chaque unité (composant) est typée par ses interfaces, qui spécifient les services requis et fournis par ce composant.

1. European Conference on Object-Oriented Programming, 20th edition. July 3-7, 2006. Nantes (France)

Note 2.

Dans la suite de cette thèse, nous considérons un composant comme une unité de composition, qui spécifie de manière explicite ses services fournis et/ou requis à travers des interfaces contractuelles.

2.2.1.2 Vue externe d'un composant

Une vision métaphorique perçoit un composant ainsi : *« a component is a static abstraction with plugs »* [ND95]. Le terme *plugs* dans cette définition peut être traduit par "prises". Ces dernières constituent les uniques points d'accès d'un composant permettant son assemblage au même titre que les pattes d'un composant électronique. Même si le terme « point d'accès » n'est pas d'un usage commun dans la communauté CBSE (ni le terme « prise » d'ailleurs), nous allons toutefois, dans un premier temps, employer ce terme afin de ne pas introduire d'ambiguïtés avec la terminologie existante.

Les points d'accès d'un composant contribuent à renforcer l'encapsulation du composant auquel on ne peut accéder que via l'un de ses points d'accès. Ils augmentent également l'indépendance des composants par rapport à leur environnement (d'autres composants) en rendant explicites les interactions avec celui-ci. À travers ses points d'accès, un composant fournit une ou plusieurs fonctionnalités.

Les fonctionnalités représentent les traitements que peuvent accomplir des composants. Dans certains modèles, un composant n'a qu'une seule fonctionnalité. Toutefois, dans la plupart des modèles actuels, un composant en possède plusieurs. Selon le modèle de composants considéré, ces fonctionnalités sont exposées à travers différentes entités telles que : les ports, les interfaces ou les services.

La différence entre ces différentes entités n'est, dans le cas général, pas très claire. Cela est essentiellement dû au manque d'un standard unifiant les modèles de composants, ainsi on peut trouver la même entité jouant des rôles différents d'un modèle à un autre. Nous détaillerons ce point dans la section 2.5.1.

L'étude de la représentation des points d'accès d'un composant dans les différents modèles nous a permis de distinguer deux problématiques dans lesquelles ils interviennent :

- la première est la description des composants : les points d'accès de composants sont perçus comme des descriptions (syntaxiques, fonctionnelles, comportementales, *etc.*) permettant, par exemple, la vérification des assemblages, la validation des utilisations des composants, ou le raffinement vers le code exécutable, *etc.* Les notions d'interface, de protocole et de contrat sont souvent confondues pour décrire cet aspect ;
- la seconde est l'assemblage des composants : ici, les points d'accès de composants sont perçus comme les points de connexion des composants et comme le support de communication. Le concept de port est généralement utilisé pour représenter cette facette.

Notion d'interface

De façon générale, les interfaces sont un support de description des composants permettant de décrire l'ensemble des fonctionnalités offertes et requises par le composant, souvent sous forme de signatures de méthodes. L'interface est aussi un moyen d'expression des dépendances entre un composant et le monde extérieur.

Cependant, l'interface telle que décrite ci-dessus ne permet pas de préciser complètement le comportement du composant. En effet, il est possible d'enrichir une interface par l'ajout de certaines propriétés sur les aspects dynamiques et de contraintes liées aux composants et à leur assemblage. Cet enrichissement permet alors de faciliter la vérification et la prédiction de propriétés d'assemblages de composants. **Les protocoles** peuvent ainsi être utilisés pour permettre de spécifier des contraintes sur l'utilisation des composants. Par exemple, un composant dédié à des communications réseaux offre les trois fonctionnalités suivantes : ouvrir(adr) (ouvre une connexion réseau à l'adresse spécifiée par le paramètre (adr), envoyer(data) (envoie les données data à travers la connexion) et fermer (ferme la connexion). Un protocole pour un tel composant permet de spécifier que l'ordre d'invocation de ces trois fonctionnalités pour une utilisation valide est : ouvrir (appelé une seule fois), puis envoyer (autant de fois que nécessaire) et fermer (une seule fois), ce qui peut s'exprimer par $ouvrir.envoyer^*.fermer$ en utilisant le formalisme des expressions rationnelles. Les interfaces peuvent également être enrichies en utilisant la notion de contrat. **Les contrats** permettent de décrire de manière explicite les obligations et les dépendances d'un composant (les contrats seront étudiés plus en détail dans la section 2.3).

Enfin, il est important qu'une interface soit définie de manière indépendante de toute implantation. Il est ainsi possible, d'une part, de fournir plusieurs implantations pour un même type de composant et, d'autre part, de pouvoir substituer une implantation par une autre mieux adaptée.

Notion de port

Dans le domaine des réseaux et des systèmes répartis, on définit un port comme un point d'accès au service rendu par un logiciel "communicant" (par exemple, port TCP [1]). La communauté CBSE a repris cette terminologie dans une acceptation très voisine. Un port de composant est un point d'accès à l'ensemble (ou une partie) des fonctionnalités impliquées (fournies ou requises) dans ce composant. Un composant ne peut communiquer que via ses ports. Les fonctionnalités d'un port sont réunies selon un point de vue cohérent retenu par le concepteur d'un composant.

Dans la plupart des modèles supportant le concept de port, on distingue les ports requis (définissant les points de vue que le composant peut avoir sur des composants externes) et les ports fournis (définissant les points de vue que les composants externes peuvent avoir sur ce composant). De même, lorsque l'on accède à un composant via l'un de ses ports, ce dernier est le garant d'une politique de sécurité, c'est-à-dire que les composants clients ne pourront utiliser que les services disponibles via ce port. L'idée sous-jacente à la création d'un port est la suivantes :

1. Transmission Control Protocol

si une fonctionnalité appartenant à un port d'un composant est utilisée par un autre composant, alors il est très probable qu'une autre fonctionnalité du même port soit aussi impliquée dans la coopération entre les deux composants.

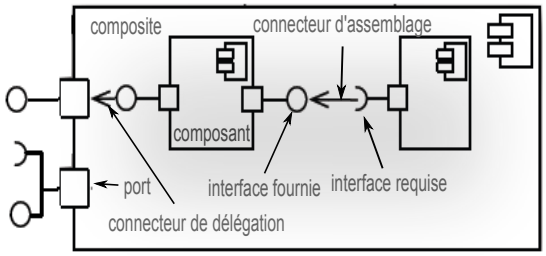


FIGURE 2.2 – Exemple d'une vue externe d'un composant UML 2.0

Par exemple, un composant UML est vu comme une boîte noire [Gro03]. La partie interne d'un composant n'est accessible qu'au travers de ses interfaces. Une interface définit un type. Elle contient un ensemble de méthodes et/ou de contraintes. Une interface (fournie ou requise) peut être attachée à un port, lui-même attaché à un composant (cf. figure 2.2). Un port d'un composant UML peut contenir des interfaces requises ou fournies. Ce type de port est très utile pour décrire l'interaction d'un composant avec son environnement dans le cas de service complexe. Un port peut se voir attacher un comportement pour décrire les interactions qui doivent se produire sur le port.

2.2.1.3 Services d'un composant

Le terme *service* est souvent employé pour désigner une fonctionnalité. Dans le contexte des systèmes répartis, l'infrastructure proposée par les intergiciels [Ber96] offre de nombreux services pour rendre interopérables des systèmes hétérogènes (des systèmes s'exécutant sur des environnements matériels et logiciels différents). Chaque service est normalisé et les autres systèmes peuvent y accéder uniformément. Par exemple, l'intergiciel CORBA [Vin97] comporte dix-sept services : un service de gestion de cycle de vie pour créer/supprimer/copier/déplacer des objets, un service d'annuaire pour nommer/retrouver des objets, un service transactionnel pour assurer l'atomicité d'un ensemble d'opérations, un service d'horloge pour synchroniser des systèmes, *etc.*

Dans les architectures orientées services (SOA [1]) [BH06], un service est un mécanisme offrant à un client des fonctionnalités proposées par un fournisseur (on parle aussi de services métiers [Erl05]). Le fournisseur ne connaît pas forcément les futurs clients et, de même, le client ne sait pas comment est réalisé le service attendu. Le client peut éventuellement utiliser le service d'une manière non prévue par le fournisseur. L'accès au service se fait via une interface contenant les détails de ses fonctionnalités afin de déterminer si le service est approprié aux besoins de son client ou non.

1. Service-Oriented-Architecture

Un service correspond donc à une entité logicielle fournissant une ou plusieurs fonctionnalités et qui est mise à disposition par un fournisseur, en vue d'une utilisation par un tiers (un autre service ou une application) considéré comme le client du service. Dans la littérature, il y a plusieurs définitions du concept de service, parmi elles :

A service is software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. Services should ideally be governed by declarative policies and thus support a dynamicaly reconfiguration architectural style. [AGA+08]

Cette définition spécifie qu'un service doit posséder une description accessible depuis l'extérieur. Cette description permet aux clients (utilisateurs) du service de rechercher, de trouver et d'invoquer le service correspondant à cette description. Ainsi, cette définition décrit le modèle d'interaction existant entre les fournisseurs et les clients du service. Elle précise aussi qu'un service peut disposer d'un ensemble de politiques (propriétés) exprimées de façon déclarative. Le service est donc un concept fondamental pour la définition explicite des contrats entre clients et fournisseurs (cf. section 2.3.5).

En résumé, un service possède les caractéristiques suivantes :

- une description rendue accessible par des fournisseurs pour d'éventuels clients ;
- sa description peut contenir l'ensemble des fonctionnalités qu'il fournit, ainsi que ses propriétés ;
- sa description ne contient pas les détails liés à la technologie utilisée, tel que le langage de programmation.

Dans le contexte des CBSE [CL00], un service est fourni par un composant. Un composant peut également requérir d'autres services pour réaliser ceux qu'il fournit. Dans ce cas, il est à la fois fournisseur et client.

La manière dont un service est rattaché à un composant diffère suivant les auteurs :

(a) un service correspond à une interface exposée par un composant. Cette interface liste les opérations du service. Le service correspond alors à un ensemble cohérent d'opérations réalisées par un composant et exposées dans une interface [Szy97, OKS05a] ;

(b) une interface expose des services, chaque service correspond à une fonctionnalité [LW07].

(c) un service a plusieurs interfaces, chaque interface liste les opérations du service. Il est ainsi possible d'accéder de plusieurs manières au service. Ceci est notamment utile lorsque le service évolue : chaque interface correspond alors à une version « historique » du service.

(d) un service est réalisé par plusieurs composants (vision SOA [BH06, Pap03]).

On distingue les services requis et les services offerts. Le service offert d'un composant désigne une fonctionnalité définie à l'intérieur de ce même composant et offerte aux autres composants pour qu'ils puissent l'appeler. Tandis que le service requis d'un composant, exprime le besoin d'une fonctionnalité nécessaire au fonctionnement de ce composant (invoqué dans le comportement de ses services fournis)

et fournie par d'autres composants.

Le nombre de services qu'un composant fournit est aussi une caractéristique du composant. Si un composant n'offre qu'un seul service, alors ce composant aura une granularité faible : de ce fait, le système global devra comporter de nombreux composants pour assurer toutes ses fonctionnalités (c'est-à-dire autant de composants que de fonctionnalités) et il sera alors plus difficile à maîtriser. En revanche, autoriser un composant à offrir plusieurs services permet d'agglomérer les services, on parle alors de composant *multi-services*. Le composant propose donc un point d'accès centralisé à plusieurs services permettant d'appréhender plus facilement la complexité du système global. Cette problématique rejoint celle de la composition que nous détaillons ci-après.

> ✎ **Note 3.**
>
> Dans cette thèse, nous considérons des composants multi-services (vision (b) c'est-à-dire que l'interface expose plusieurs services). Nous distinguons également les services des opérations par une granularité plus importante permettant de modéliser des fonctionnalités plus complexes.

2.2.2 Composition

Comme nous l'avons vu dans la section précédente, un composant peut ne pas être directement utilisable, c'est-à-dire que l'appel d'une de ses fonctionnalités ne fonctionnera pas si le composant n'est pas assemblé au préalable, car il requiert une fonctionnalité offerte par un autre composant. Nous distinguons deux mécanismes pour connecter les composants : l'assemblage (composition horizontale) et le composite (composition verticale).

L'assemblage est une notion très utilisée dans la plupart des domaines de l'ingénierie comme le génie civil ou encore le génie électrique. Dans ces domaines, il correspond à la construction d'édifices pour une utilisation quelconque par l'humain. Le terme *assemblage*, dans le contexte du CBSE, désigne un ensemble de liaisons entre deux ou plusieurs composants formant l'application (ou partie de l'application). Ces liaisons permettent de satisfaire les dépendances des composants exprimées via leurs points d'accès (cf. section 2.2.1.2) en utilisant les fonctionnalités fournies par d'autres composants (relation de clientèle). La notion d'assemblage est importante pour déterminer si les composants sont bien reliés, si leurs points d'accès sont compatibles et si la combinaison de leurs sémantiques (comportements) aboutit au comportement global désiré.

La composition verticale de composants consiste à créer un nouveau composant encapsulant un (ou plusieurs) assemblage(s) avec éventuellement la promotion (ou délégation) de certains services de ses sous-composants (relation d'inclusion ou d'agrégation forte). C'est le cas d'un composant qui offre un service réalisé par un de ses composants internes, ou dont un composant interne requiert un service qu'il est obligé de faire figurer dans ses services requis. Les composants ainsi créés sont dits *composites* car ils sont eux-mêmes constitués de composants plus élémentaires.

> ✎ **Note 4.**
>
> La distinction entre composition horizontale et composition verticale relève de la nature des liaisons entre les entités exposées (points d'accès) par les composants. La première relie des points d'accès de nature différente (requis/offert) ; on parle alors de relation de clientèle dans un assemblage. La seconde relie des points d'accès de même nature (requis/requis ou offert/offert) ; on parle plutôt de délégation dans un composite.

Nous détaillons ces deux notions dans les sections suivantes.

2.2.2.1 Assemblage *vs.* architecture logicielle

Bien qu'issus de communautés au départ peu liées, les travaux relatifs aux modèles de composants et aux architectures logicielles (notamment les ADLs[1]) sont complémentaires et tendent à se rapprocher. Les premiers visent à fournir les briques de base pour la construction des logiciels réutilisables. Les seconds se sont concentrés sur la description de la structure des systèmes en utilisant ces briques ainsi que leurs interactions.

Un ADL est, selon Medvidovic et Taylor [MT00], un langage qui offre des moyens pour la définition explicite et la modélisation de l'architecture d'un système logiciel, comprenant au minimum : des composants, des connecteurs, et des configurations architecturales.

La notion de *composant* a pris de nombreuses formes dans les différents ADLs, souvent plus générales que la seule notion de composant logiciel (cf. section 2.2.1) car le domaine pris en compte était généralement plus large que celui du logiciel. Ainsi, certaines approches autorisent par exemple la définition de composant matériel.

Le *connecteur* correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions.

> ✎ **Note 5.**
>
> Nous ne considérons pas, ici, les connecteurs comme des entités de première classe. En revanche, les liaisons définies dans les compositions (horizontales ou verticales) peuvent être considérées comme des connecteurs simples.

Dans les ADLs, les composants et les connecteurs peuvent être assemblés à partir de leurs interfaces pour former une *configuration*. Celle-ci décrit l'ensemble des composants logiciels nécessaires au fonctionnement d'une application, ainsi que leurs interactions.

1. Architecture Description Languages

> ✎ **Note 6.**
>
> La notion de configuration dans les ADLs s'apparente à la notion d'assemblage dans les modèles de composants.

2.2.2.2 Composite

Dans les approches à composants, un *composite* est un composant constitué d'autres composants appelés sous-composants (ou composants internes). Ce concept est à la base de la relation de composition verticale. Nous rappelons que l'expression des services requis d'un composant permet son découplage et augmente ainsi son potentiel de réutilisation. Cette technique d'extraction des dépendances du code source sous la forme de services requis est parfois appelée *factoring out* [SC00]. Bien que cette technique permette un meilleur découplage, elle pose aussi des problèmes de passage à l'échelle et de réutilisation. Les composites constituent notamment une réponse à ces besoins, à savoir :

- encapsuler plusieurs composants et leur assemblage afin de masquer certains détails dans une architecture logicielle,
- réutiliser directement des assemblages de composants.

Nous pensons qu'un modèle de composants doit permettre la manipulation (définition, réutilisation, remplacement, maintenance, etc.) non seulement des composants mais également de leurs assemblages. Pour cela, comme nous l'avons déjà mentionné dans la section précédente, les ADLs [MT00] ont été des précurseurs en proposant le concept de configuration. Toutefois, contrairement à une configuration, un composite est un composant en tant que tel. Les modèles supportant le concept de composite sont dits *hiérarchiques* car un composite peut être décomposé en un assemblage de sous-composants interconnectés, chaque sous-composant pouvant être un composite ou un composant de base.

En effet, considérer les composites comme des composants permet :

- de les mettre sur étagère et de les réutiliser (selon le principe de COTS[1]) ;
- de les doter de services requis permettant ainsi d'encapsuler des architectures « partielles » qui pourront être raffinées via le mécanisme d'assemblage ;
- de rendre les architectures complexes plus compréhensibles puisqu'elles peuvent être examinées à différents niveaux de granularité, suivant que l'on détaille ou non le contenu d'un composite.

La conception d'un composite nécessite de faire un choix entre ce qui doit être externalisé (via des interfaces requises) afin de garder un fort potentiel de réutilisation et ce qui doit être internalisé en utilisant des sous-composants afin de masquer les détails de fonctionnement. L'utilisation systématique de composites sans externalisation des besoins ne permet pas de définir des composants réutilisables. L'objectif est que tous les composants (et même les sous-composants) soient mis sur étagère afin d'être réutilisés. Ce choix est motivé par le fait qu'un composant ne doit pas uniquement être conçu comme un sous-composant encapsulé dans un composite donné

1. *Component-Off-The-Shelf* : acronyme utilisé pour désigner des composants prêts à l'emploi.

mais comme un composant mis sur étagère et éventuellement utilisé en tant que sous-composant. Les sous-composants sont inaccessibles (voire invisibles) en dehors de leur implantation (c'est-à-dire perçus comme des boîtes noires).

La notion de composite est très utile car elle permet d'abstraire les détails d'implantation des composants internes. Ainsi, elle permet également de décrire le système à un haut niveau d'abstraction pouvant être potentiellement compris par des personnes de différents niveaux d'expertise et de connaissances techniques. Cela permet de favoriser la communication entre les différents acteurs impliqués dans le développement du système global.

2.2.3 Cycle de développement à base de composants

De plus en plus de travaux visent à définir des méthodes de développement adaptées à l'approche à composants [DW99, APRS01]. La définition de ces nouvelles méthodes de développement a fait émerger de nouveaux acteurs tels que le développeur de composants, l'architecte d'applications, ou encore le déployeur de systèmes [OZA07b]. Deux acteurs nous intéressent plus particulièrement : le développeur et l'architecte. Leurs rôles complémentaires sont au centre du processus de développement par composants (cf. Figure 2.3).

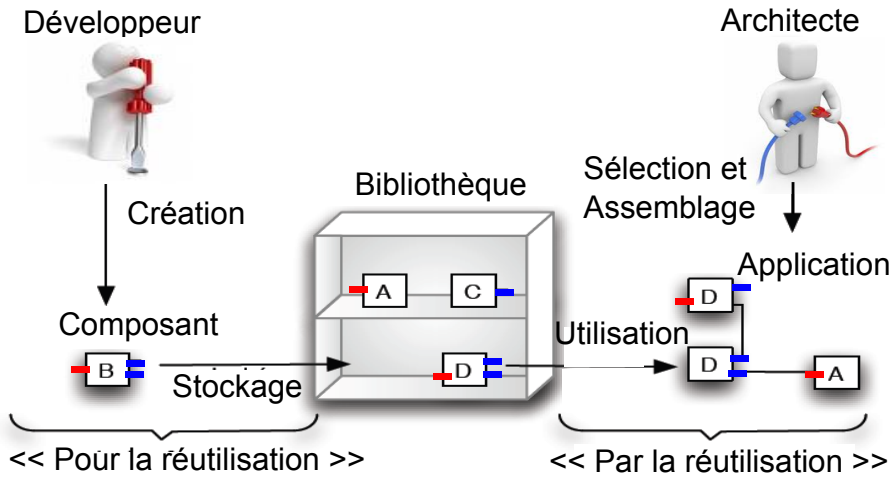


FIGURE 2.3 – Principe du développement des systèmes à base de composants.

Le développeur a pour objectif la création (spécification et implantation) de composants complètement indépendants de l'application pour laquelle ils sont conçus, afin d'être éventuellement réutilisés dans d'autres applications. À l'inverse, l'architecte construit une application ; il choisit donc des composants que des développeurs ont mis sur étagère, qu'il adapte, si nécessaire, pour les intégrer à son application. En ce sens, la conception faite par le développeur s'oriente vers une perspective de réutilisation (*« design for reuse »*) alors que celle de l'architecte se fait essentiellement par la réutilisation (*« design by reuse »*) [OKS05b].

La construction d'applications à base de composants telle qu'elle est schématisée dans la figure 2.4 montre le lien entre ces deux activités fondamentales et le reste des

phases de développement à base de composants tels que le déploiement, l'installation, l'exécution et l'administration des applications. En effet, il est indispensable à chaque étape du développement et dès qu'un besoin apparaît, de systématiquement rechercher l'existance d'un composant déjà disponible, capable de le satisfaire tout en s'intégrant à moindre coût dans l'architecture logicielle en cours d'élaboration. Il s'agit ici d'adopter, au sein de la philosophie de développement, des réflexes favorisant véritablement la réutilisation. Ainsi, le développement à base de composants logiciels diffère largement d'un développement "classique" [CCL06].

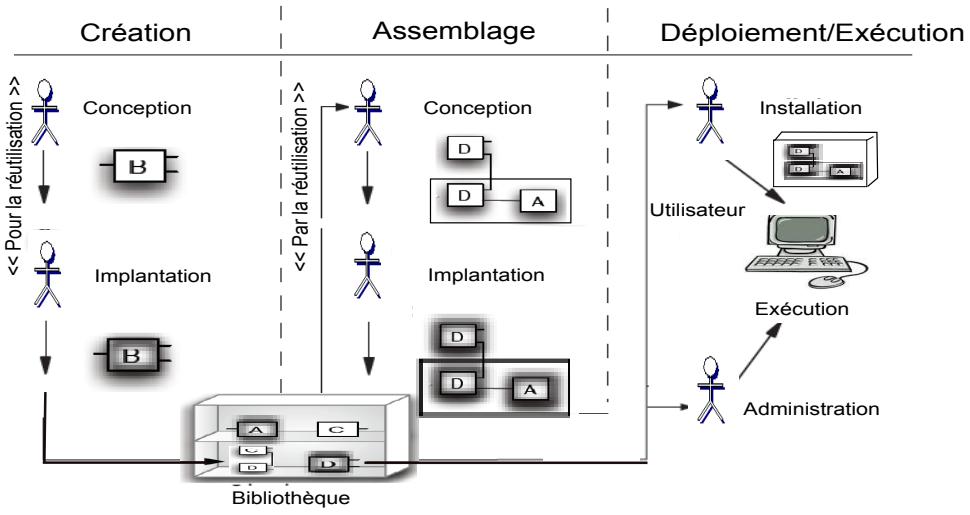


FIGURE 2.4 – Cycle de développement simplifié des systèmes à base de composants.

Création : cette phase consiste à définir et à implanter les interfaces ainsi que les composants eux-mêmes : services offerts, services requis, pré- et post-conditions, protocoles, etc. La réalisation (implantation) doit se faire en respectant les contraintes spécifiées dans les interfaces. Une fois ces éléments définis et implantés, ils peuvent être livrés aux tiers et/ou mis dans des bibliothèques.

Assemblage : la phase d'assemblage consiste à relier les composants pour former une (ou une partie d'une) application. Un assemblage reflète l'architecture d'une application et peut être lui-même empaqueté dans un composant pour être assemblé à son tour, mis sur étagère et utilisable par des tiers. De plus, un assemblage contient la configuration [1] de chaque composant et celle des liaisons qui les relient.

Déploiement : cette phase permet d'extraire le composant de son paquetage pour l'installer dans son environnement. Pour mener à bien cette installation, il faut disposer, ce qui n'est pas souvent le cas, de structures d'accueil permettant d'une part d'instancier des composants et de les installer aux endroits désirés, et d'autre part, de les exécuter.

1. Au sens de paramétrage d'un composant. À ne pas confondre avec le concept de *configuration* dans les architectures logicielles vu dans la section 2.2.2.

Exécution et utilisation : la phase d'exécution correspond à l'exploitation effective du composant. Les acteurs potentiels de cette phase sont, d'une part, les administrateurs qui vont installer et maintenir les applications et, d'autre part, les utilisateurs finaux qui vont exploiter les services proposés.

2.2.4 Critique des approches à composants actuelles

Bien que l'approche CBSE permette, comme nous avons pu le constater dans les sections précédentes, de mieux appréhender les systèmes à grande échelle, nous pouvons toutefois légitimement nous poser les questions suivantes : « Qu'en est-il de leur vérification ? Sont-ils sûrs, fiables, corrects ? ». La phase de vérification est désormais reconnue comme essentielle dans le cycle de développement des logiciels (en particulier les logiciels critiques). Elle intervient dans les différentes étapes du cycle de développement (phases d'analyse, de conception et d'intégration *etc.*) permettant ainsi d'améliorer la qualité des logiciels.

Lors de cette étude, nous avons pu identifier deux manques considérables : le premier au niveau de la vérification des systèmes à base de composants de façon générale, et en particulier la vérification des propriétés fonctionnelles (c'est-à-dire, s'assurer que le système se comporte exactement comme il a été spécifié) ; et le second dans le fait que même les rares approches à base de composants qui traitent de la vérification ne s'y intéressent qu'après la phase d'implantation (en *runtime*) et ne proposent pas suffisamment de moyens et de techniques d'analyse dans les phases antérieures (*design time*).

Cependant, nous sommes convaincus que l'apport des composants en terme de séparation claire entre leur savoir-faire (fonctionnalités) et leurs besoins d'une part, et le support de composition d'autre part, facilite largement leur vérification modulaire. En effet, ces éléments constituent les concepts de base du paradigme de contrat basé sur la notion d'*hypothèse-garantie* [SDW93], un paradigme qui est devenu fondamental dans la conception compositionnelle des systèmes informatiques [Mey03]. Nous avons ainsi défini un cadre de travail, dans lequel les contrats sont utilisés pour vérifier la correction des hypothèses faites sur l'environnement des composants, et pour assurer à l'environnement les garanties exprimées par les composants. Dans la section suivante, nous détaillons le paradigme de contrats.

> ✎ **Note 7.**
>
> Dans cette thèse, nous nous intéressons plus particulièrement à la création et à l'assemblage des composants ainsi qu'à la vérification de leur correction au *design time*. Le déploiement et l'exécution de ces derniers ne seront donc pas traités.

2.3 Les contrats

Un contrat se définit comme une convention formelle ou informelle, passée entre deux parties ou davantage, ayant pour objet l'établissement d'obligations à la charge ou au bénéfice de chacune de ces parties. Les dispositions d'un contrat sont appelées clauses ou stipulations. [1]

Dans cette section, nous étudions les différentes approches de développement basées sur les contrats qui ont été proposées initialement dans le contexte des objets, puis des services et des composants. Nous nous intéressons notamment aux formes de contrats et à certaines techniques de spécification et de vérification dans ces trois paradigmes.

2.3.1 Avènement des contrats logiciels

La conception par contrats (DbC pour *Design by Contract*) [Mey92] est une méthode de conception logicielle. Elle est basée sur la théorie des types de données abstraits [Rua84] et sur la métaphore du contrat au sens juridique cité plus haut. Le DbC met en avant l'intérêt de spécifier précisément les interfaces d'un module logiciel en termes de pré-conditions, post-conditions et invariants.

Bien que le terme DbC ait été proposé par Bertrand Meyer en 1992 [Mey92], des travaux sur les contrats remontent à Floyd [Flo67] et Hoare [Hoa69]. Ces travaux montrent l'intérêt de l'utilisation des assertions (qui sont parfois appelées contrats) dans la spécification et la correction de programmes. Une assertion est définie comme une expression booléenne qui doit être vraie pour qu'un programme spécifié s'exécute correctement. Cependant, d'un point de vue conceptuel, les contrats vont au-delà des mécanismes d'assertions. En effet, le paradigme de contrat a été très étudié par la communauté de génie logiciel [Mey92], face à des modules logiciels dont le fonctionnement global est devenu trop complexe à comprendre aisément. Les contrats sont utilisés comme un sous-ensemble de la spécification du système permettant de fixer un certain nombre d'invariants que le (ou une partie du) système s'engage à respecter ou que l'environnement de ce dernier doit respecter également.

2.3.2 Avantage de l'approche par contrat

L'utilisation des concepts du DbC dans le développement d'une application occasionne de très nombreux bénéfices [Rap02]. En voici quelques uns :

Séparation des préoccupations

L'expression explicite d'un certain nombre d'obligations relatives au bon fonctionnement de différents modules d'une application permet de clarifier le code de cette dernière. Ce code contient alors uniquement le code fonctionnel lié aux méthodes, et n'est pas envahi par des parties de code vérifiant la correction des paramètres envoyés ou retournés au cours d'une interaction. Par exemple, une méthode

1. http ://fr.wikipedia.org/wiki/Contrat

n'a pas à vérifier sa pré-condition ; et, après un appel de méthode, son appelant n'a pas à vérifier la post-condition non plus.

Possibilité d'outillage

L'approche basée sur les contrats possède un lien très fort avec les méthodes formelles permettant de vérifier la correction d'un module logiciel. Le contrat lui-même constitue une forme de spécification. Mais, contrairement aux méthodes formelles, on ne va pas chercher à montrer explicitement que la spécification est bien réalisée par le module logiciel. Cette partie est laissée à la discrétion de son client et de son fournisseur. Néanmoins, on met souvent en place des mécanismes de vérification et de tests pendant l'exécution afin de vérifier leur validité.

Grâce à une formalisation précise des langages de contrats, il devient donc possible de réduire le fossé entre le langage de spécification et le langage d'implantation. Cela facilite largement la possibilité d'outiller convenablement l'approche afin de garantir statiquement certaines propriétés, raffiner les spécifications et générer du code respectant les contrats auparavant définis.

Documentation

Les contrats peuvent également être utilisés pour la génération automatique de la documentation à partir du code et de ses commentaires, tout comme le permettent des outils disponibles dans de nombreux langages. Par exemple, l'outil Jmldoc [BCC+05] permet de générer une Javadoc améliorée incluant des informations tirées des contrats exprimés en JML (*Java Modeling Language*) [LRR+00].

2.3.3 Techniques d'intégration de contrats

Diverses études [Kra98, LRR+00, PK02] mettent en relief des traits caractéristiques des mises en œuvre des contrats au niveau des langages.

La technique la plus simple consiste à créer un préprocesseur qui produit du code source correspondant aux contrats et l'intègre dans le code source du programme (ou de la spécification). Cette méthode a l'avantage de séparer le code décrivant le comportement de celui des spécifications. Cependant, elle présente l'inconvénient de modifier le code source, ainsi, les numéros de ligne du code source ne sont plus utilisables tant à la compilation (erreurs), qu'au débuggage, *etc.*

Une alternative consiste à étendre la grammaire du langage de base. Les contrats sont alors traités par le compilateur de la même manière que le reste du programme (ou spécification). L'intérêt de cette approche réside en l'intégration homogène dans la grammaire du langage de base permettant la prise en compte des contrats dans les outils associés (compilateur, *debugger*, *etc.*).

Enfin, une technique basée sur la métaprogrammation et la réflexivité des langages peut également être utilisée. Les contrats peuvent alors se présenter sous une forme exécutable distincte du programme. Leur mise en œuvre repose sur l'étude réflexive, en cours d'exécution, des objets utilisés par le programme [AC01]. Cette

approche évite l'utilisation d'outils de compilation ou de précompilation particuliers, ainsi que la modification du code source, mais suppose la prise en compte des contrats par l'environnement d'exécution.

L'intégration du paradigme de contrat a pris de nombreuses formes selon le contexte d'application (objets, services, composants, *etc.*). Dans la suite, nous verrons plus en détail la mise en œuvre des contrats dans les approches à objets, services et composants.

2.3.4 Contrats d'objets

La conception par contrat s'est appliquée d'abord à la programmation orientée objet. Dans ce cadre, une application est constituée d'un ensemble d'objets qui collaborent (comme peut le décrire le diagramme de collaboration d'UML [Gro03]) en s'échangeant des messages, des appels de méthodes et leurs retours.

Les contrats appliqués aux objets reposent, en général, sur la mise en œuvre d'assertions exprimant les obligations de chacune des parties. Les obligations du client sont nommées les **pré-conditions** ; elles sont vérifiées à l'entrée de la méthode qu'elles contraignent. Les obligations du fournisseur sont appelées les **post-conditions** ; elles sont vérifiées à la fin de l'exécution de la méthode. Enfin, des **invariants** peuvent également être utilisés pour contraindre les états observables d'une instance de classe, permettant ainsi d'assurer sa cohérence interne. Le travail le plus connu dans ce domaine est probablement celui autour d'Eiffel [1] [Mey92]. Dans le contexte d'Eiffel, ces assertions sont notées "require" (pré-conditions), "ensure" (post-conditions) et "invariant" (invariants). Un exemple de spécification en Eiffel est présenté dans le listing 2.1.

Listing 2.1 – Exemple de spécification en Eiffel.

```
class interface
ACCOUNT
feature                    ----- Accès
balance: INTEGER           ----- Solde courant
deposit_count: INTEGER     ----- Nombre de dépôts depuis l'ouverture
deposit( sum: INTEGER)     ----- Créditer le compte

----- pré-condition
require
non_negative: sum>= 0

----- post-condition
ensure
one_more_deposit: deposit_count = old deposit_count + 1
updated: balance = old balance + sum

----- invariant
invariant
consistent_balance: balance= all_deposit.total
end -- class interface ACCOUNT
```

1. Le nom du langage provient de l'ingénieur Français Gustave Eiffel, concepteur bien connu de la tour Eiffel, qui a été construite dans les délais et en respectant le budget, ce qui est le but avoué du langage Eiffel pour les projets logiciels importants. Le nom lui-même du langage est donc un clin d'œil à l'objectif de respect de saines pratiques lors du développement de logiciels.

Si un contrat n'est pas satisfait, ses éléments déterminent l'endroit où se produit la violation associée. Par exemple, une violation de pré-condition génère une exception dans le client, une violation de post-condition génère une exception dans l'opération et une violation d'un invariant génère une exception dans l'opération de la classe ayant causé la rupture de l'invariant.

Selon l'approche, les assertions peuvent intervenir à différents moments du cycle de vie du système. On distingue les assertions exécutables et les assertions en conception que nous détaillons ci-dessous.

Assertions exécutables

De nombreux travaux ont proposé la mise en œuvre des contrats dans les langages de programmation, notamment autour de Java (le lecteur peut se reporter à [PK02] pour un tour d'horizon de ces travaux). Dans le cadre des extensions de Java, on retient principalement iContract [Kra98] qui fut le premier prototype fonctionnel. Il était basé sur un préprocesseur (source vers source) et utilisait les annotations javadoc pour intégrer les assertions aux fichiers sources Java. On retiendra aussi JML [LRR+00] qui insère des commentaires dans un code source Java pour lui associer un modèle algébrique abstrait développé séparément. Cela permet de réutiliser des outils de preuve indépendamment des langages traités. JML présente également la particularité de combiner des vérifications dynamiques (par l'évaluation à l'exécution des assertions) et statiques. Différents outils d'analyse ont été greffés à la machinerie JML, comme ESC/Java, JACK et Key [BCC+05], *etc.*

Assertions en conception

L'approche de conception par contrat était, au départ, fortement liée à l'activité de conception orientée objet. En effet, la première repose sur l'expression des conditions de collaboration entre objets. Or l'expression de ces collaborations est à la base de la distinction du contour des objets par les responsabilités qu'elle amène à faire apparaître. OCL est apparu en 1997 avec la version 1.1 d'UML [OMG05]. C'est un langage de formalisation de contraintes d'une modélisation UML qui se veut accessible aux non spécialistes des techniques formelles. Il propose un ensemble de fonctionnalités consistant en l'expression d'invariants de modélisation. Une contrainte OCL se rattache au modèle UML par l'intermédiaire d'un contexte. Un contexte spécifie une entité (classe, types, méthode *etc.*) sur laquelle les expressions OCL doivent être interprétées. Ce contexte est complété d'un stéréotype : *inv, pre, post* décrivant sa portée temporelle ; *inv* est associé aux classes, types et stéréotype, alors que *pre*, *post* s'appliquent aux méthodes ou opérations des classes. Le stéréotype *inv* s'étend sur la durée de vie de l'entité, alors que *pre* (resp. *post*) définit l'instant précédent (resp. succédant) l'exécution d'une méthode. OCL permet d'associer à ce contexte des contraintes sous forme d'expressions booléennes dont la validité devra être assurée sur celui-ci. Afin d'être accessible aux développeurs, la syntaxe des expressions OCL est inspirée du langage smalltalk qui est reconnu pour sa simplicité.

2.3.5 Contrats de services

Dans le domaine des SOA [BH06], le contrat prend une importance particulière [BZ08]. En effet, l'utilisation de services s'intègre dans des processus-métiers qui sont contraints par des critères tant fonctionnels que non-fonctionnels. Compte tenu de l'importance de tels services (services électroniques, e-business), il est nécessaire de spécifier les relations qui existent entre fournisseurs et clients de services ou entre différents fournisseurs qui coopèrent. Une forme de contrats (appelés SLA pour *Service Level Agreement*) a été proposée pour expliciter et identifier les besoins et les engagements des différentes parties impliquées dans la réalisation des services. Ces contrats sont généralement bilatéraux et traduisent ainsi de façon informelle le fait que, d'un côté, les fournisseurs s'engagent à fournir des services aux qualités spécifiées à la condition que les directives de l'utilisation des services soient suivies, et que d'un autre côté, les clients acceptent de respecter les directives d'utilisation pour pouvoir alors bénéficier des qualités de service garanties.

Les SLA : Service Level Agreement

Un *Service Level Agreement*, que l'on pourrait traduire en français par « accord de niveau de service » ou « contrat de niveau de service », est un document qui définit la qualité de service requise entre un prestataire et un client [BBC+98].

Historiquement, les SLA sont utilisés depuis la fin des années 1980 par les opérateurs téléphoniques (pour le téléphone fixe) dans leurs contrats avec les entreprises. Plus récemment, les départements informatiques de certaines grandes entreprises ont repris l'idée et utilisent les SLA avec leurs clients - généralement, des utilisateurs d'autres départements au sein de l'entreprise - pour permettre une comparaison entre la qualité de service fournie et celle promise et donc pour potentiellement envisager de délocaliser le service informatique en cas de qualité insuffisante.

Dans le contexte des SOA, un SLA est la formalisation d'un accord négocié entre deux parties. C'est un contrat entre client et fournisseur, ou entre fournisseurs. Il met par écrit l'attente des parties au niveau des services, des priorités, des responsabilités, des garanties, et donc de ce que l'on pourrait finalement définir comme le « niveau de service ». Par exemple, il peut permettre de spécifier les niveaux de disponibilité, de performance, de priorité ou de tout autre attribut du service en question, tel que la facturation voire les pénalités (financières ou autres) en cas de manquement au SLA [BSC99]. Un SLA est donc un contrat (ou la partie du contrat de service) dans lequel on formalise la qualité du service en question. Dans la pratique, le terme SLA est quelquefois utilisé en référence au temps de délivrance et/ou à la performance (du service) tel que défini dans le contrat.

Les contenus des SLA sont généralement structurés selon les trois classes d'informations suivantes :

1. des informations légales décrivant les différentes parties en présence (noms, adresses, signatures, etc.) ainsi que les dédommagements en cas de non satisfaction des termes techniques des contrats à la fois pour les fournisseurs et les clients,

2. des informations organisationnelles décrivant le déroulement des interactions entre les différentes parties lors de l'utilisation des services (nécessité de s'authentifier au préalable, modalités de gestion des problèmes divers, etc.),
3. des informations techniques décrivant les conditions dans lesquelles les services sont rendus. Ces conditions sont représentées sous la forme d'un niveau de service négocié, spécifiant par exemple des paramètres de qualité de service ou des valeurs garanties. Il existe différents types de contraintes de service, qui s'étendent de ceux spécifiant des aspects non-fonctionnels généraux (disponibilité, performance, etc.) à ceux s'appliquant à des utilisateurs donnés dans des contextes bien précis (l'accès à des services spécifiques pour des employés d'une même entreprise par exemple).

2.3.6 Contrats de composants

Comme nous l'avons déjà évoqué dans la définition du composant (cf. section 2.2.1), les contrats occupent une place explicite dans certaines définitions de composants, ainsi Szyperski considère qu'un composant est *"A unit of composition with contractually specified interfaces and explicit context dependencies only"* [Szy02]. On constate dans cette partie que les contrats sur les composants ne se limitent pas à la définition des interfaces des composants mais contribuent également à la garantie de leur assemblage.

Un contrat de composant est le résultat d'un accord entre deux entités du système ; la première requiert des fonctionnalités que la seconde peut offrir. Cependant, les contrats de composants diffèrent des contrats des objets par le fait que pour fournir un service, un composant requiert souvent d'autres services (fournis par d'autres composants) exprimés explicitement dans son propre contrat. En ce sens, un contrat de composant est une notion complexe qui peut se dégrader considérablement sous l'action d'un seul maillon faible. Ainsi, pour qu'un fournisseur de composant soit en position de garantir ses services par contrat, il faut que ses propres fournisseurs garantissent les leurs. L'idée centrale est que les composants logiciels doivent avoir des responsabilités envers les autres composants avec lesquels ils interagissent. Ces responsabilités sont alors fondées sur des règles formalisées entre ces entités. Des spécifications fonctionnelles, ou « contrats », sont créées pour chaque composant dans le système avant que ces composants ne soient codés. Les interactions entre les différents composants de l'application sont alors maîtrisées grâce à ces contrats.

De nombreux travaux ont été proposés pour intégrer l'approche par contrat dans le contexte du CBSE. Tout d'abord, des adaptations de l'approche par contrat appliquée aux objets (cf. section 2.3.4) ont été proposées pour certaines plates-formes à base de composants. Citons, à titre d'exemple, l'approche permettant l'intégration d'Eiffel dans la plate-forme ".Net ". Cette approche permet d'appliquer des pré/post-conditions et des invariants sur les classes des langages à objets supportés par .Net, en produisant pour chacune d'entre elles un *wrapper* en langage Eiffel [AS01a]. Néanmoins, cette approche se limite à l'expression et à l'évaluation de la conformité des composants à des spécifications sous forme d'assertions. Elle ne diffère donc que peu de sa mise en œuvre sur les objets.

D'une façon générale, dans les approches à composant, l'hétérogénéité des composants et des infrastructures rend l'évaluation des contrats plus difficile à l'exécution. Les systèmes ne sont suffisamment définis qu'après la phase d'implantation, ce qui rend leur vérification plus difficile. De plus, les spécifications des composants étant généralement plus fermées que les spécifications objet, il n'est pas toujours possible d'avoir accès aux champs caractérisant l'état du composant (approche boîte noire). Par conséquent, il faut d'abord identifier les propriétés devant être garanties par les contrats de composants avant de les vérifier. Différents travaux, comme celui de Beugnard et al. [BJPW99b] ou encore les travaux autour du modèle de composants du projet RNTL ACCORD, ont déjà exploré cette piste. Dans la section suivante, nous détaillons les différents types de propriétés devant être exprimées par des contrats de composants.

2.3.7 Contrats *vs.* propriétés

En génie logiciel, le terme contrat est souvent confondu avec les assertions (pré, post-conditions ou invariants), mais la définition des contrats va bien au-delà du mécanisme des assertions. Ces dernières ne sont qu'un mécanisme de base qui n'a que peu en commun avec les aspects méthodologiques qu'offrent les contrats au sens large.

Appeler correctement (avec la bonne syntaxe et les bons types de données échangées) un module (opération, service *etc.*) est aussi une forme de contrat pour faire fonctionner un système (application). De plus, les assertions permettent certes d'améliorer le niveau de confiance des programmes, mais cela n'est garanti que sous une hypothèse très forte assurant que l'exécution des modules est atomique ou du moins séquentielle. Néanmoins, les systèmes sont de plus en plus complexes et intrinsèquement concurrents. Des contraintes de communication et de synchronisation sont donc nécessaires pour assurer le bon déroulement de tels systèmes. Ces contraintes sont également une forme de contrat (tel module doit envoyer -ou recevoir- telle donnée à tel moment).

D'autres formes de contrat peuvent être considérées ; par exemple, le système s'engage à réagir (ou à fournir ses résultats) dans un délai précisé au préalable (éventuellement négocié avec le client), ou à garantir une certaine précision et/ou pertinence de ses résultats, *etc.* Ces exemples sont communément appelés des propriétés non-fonctionnelles (ou de QoS[1]).

Par la suite, nous expliquerons, en nous inspirant largement de la classification de Beugnard et al. [BJPW99a], ces différentes formes (ou niveaux) de contrat ainsi que les propriétés correspondant à chaque niveau.

2.3.7.1 Niveau structurel

Le niveau structurel s'intéresse plus particulièrement aux signatures (les noms des opérations, les types de paramètres d'entrées et le type de retour, *etc.*), à travers lesquelles les composants communiquent entre eux. Ainsi les principales propriétés

1. Quality of Service

exprimées à ce niveau portent sur la compatibilité structurelle entre les différents modules du système.

De manière générale, un type peut être compris comme un ensemble de valeurs sur lequel un ensemble d'opérations peuvent être réalisées correctement. L'appartenance à un type donné implique des contraintes qui vont souvent au-delà de la simple valeur dénotée, et plus particulièrement la façon dont cette valeur est stockée (nécessaire lorsque les opérations sont exécutées). Une fois les types définis, il est possible de les utiliser dans les spécifications de la forme d'un contrat (*hypothèse-garantie*) : si certaines entrées de type t_{in} sont fournies à l'opération op_{name}, alors le résultat sera de type t_{out}.

Un système de type (ou *type system*) est un ensemble de règles pour vérifier la correction des types par un processus souvent appelé contrôle de type (*type checking*). Ce processus joue un rôle très important dans la vérification de la compatibilité entre différents composants.

Le contrôle de type peut se faire statiquement (*compile-time*) et assure qu'il n'y a pas d'erreurs possibles selon les règles définies dans le système de typage. Bien évidemment, toutes les erreurs ne peuvent pas être traitées par les systèmes de typage. Par exemple, en utilisant un contrôle de type statique, il est difficile de détecter à l'avance tous les risques d'erreurs dues à la division par zéro.

D'autres propriétés peuvent être exprimées à ce niveau ; par exemple, un composant ne peut fournir une fonctionnalité que lorsqu'on lui garantit la disponibilité d'une autre qu'il ne possède pas. Ces dépendances doivent être non seulement exprimées mais également vérifiées pour garantir l'exécution du composant.

Dans le domaine des objets, les IDL (*Interface Definition Languages*) ont largement facilité l'interaction entre les objets, même lorsqu'ils sont issus de langages différents. En effet, ils permettent aux objets de décrire leurs opérations exportées ainsi que les paramètres requis, et ce d'une manière universelle.

Dans le domaine des SOA et en particulier les *web services*, le WSDL (*Web Services Description Language*) décrit une interface publique d'accès à un service web. Il s'agit d'une tentative de normalisation regroupant la description des éléments permettant de mettre en place l'accès à un service (web). Sa version 2.0 a été approuvée en 2007 et est désormais une recommandation officielle du W3C [1].

Même si les IDL sont utilisés dans CORBA [Bak97] et COM/DCOM [TEG+97], il n'existe malheureusement pas encore de langage similaire propre au paradigme du composant (au sens CBSE). Cela est essentiellement dû, comme nous l'avons vu précédemment, au manque de standard autour de la définition même du composant.

2.3.7.2 Niveau fonctionnel

Les propriétés exprimées par le niveau fonctionnel [2] s'intéressent aux valeurs réelles des données qui sont transmises entre les composants à travers les interfaces, et dont la syntaxe est spécifiée au niveau structurel. Ces propriétés permettent d'exprimer des contraintes sur la plage des valeurs admises pour un paramètre donné,

1. World Wide Web Consortium
2. Appelé « Behavioural level », à ne pas confondre avec le comportement dynamique des opérations.

ou sur la relation entre les paramètres d'un appel de méthode et sa valeur de retour. Dans les langages de contraintes, tels que fournis par Eiffel/SCOOP [Mey92], OCL [OMG05] , LSL (*Larch Shared Language*) [Hor96], JML [LRR+00], les relations entre les clients et les fournisseurs des opérations sont exprimées sous forme d'invariants, pré et post-conditions. Les travaux autour de la conception par contrats présentés dans les sections précédentes se placent à ce niveau.

Le principal problème des contrats de ce niveau est leur applicabilité aux systèmes concurrents, à cause de l'absence de moyens pour exprimer les interactions non atomiques ou pour exprimer la concurrence de façon explicite. Dans ce cas, les contrats du niveau fonctionnel ne sont pas suffisants pour assurer les propriétés fonctionnelles de tels systèmes, mis à part pour les composants primitifs contenant des opérations simples ou dans le cas où la non-interférence entre les transactions peut être garantie par construction (en imposant l'accès séquentiel aux composants), comme par exemple dans l'approche synchrone utilisée dans le contexte de la correction des applications critiques [Mor05].

Pour pallier ce problème, des propriétés liées au comportement dynamique du système (concurrence, synchronisation, communication, *etc.*) doivent être prises en compte. Ces dernières représentent des propriétés typiques exprimées par le niveau comportemental des contrats.

2.3.7.3 Niveau comportemental

Le niveau comportemental permet d'exprimer des propriétés portant sur le comportement dynamique de plusieurs modules concurrents, par exemple les dépendances entre les différents services fournis par les composants comme la séquence, le parallélisme, la synchronisation ou le choix non déterministe. Dans la suite, nous évoquons quelques techniques de description de ces propriétés.

Les algèbres de processus : ce sont des langages abstraits conçus pour décrire et analyser formellement le comportement des systèmes concurrents. Ces derniers sont modélisés par un ensemble de processus, qui exécutent des actions atomiques (briques de base du langage). Ces processus sont composés par des opérateurs spécifiques pour contraindre leurs comportements, séquentiels à la base, à s'exécuter de manière concurrente, à se synchroniser et à communiquer entre eux. Les algèbres de processus constituent une forme de contrat entre les processus impliqués dans le système global.

Les langages de référence sont CCS (*Calculus of Communicating Systems*), CSP (*Communicating Sequential Process*) [Hoa83], LOTOS (*Language Of Temporal Ordering Specification*) [BB87] et le π-calcul [Mil98].

Les algèbres de processus ont déjà connu un certain succès tant sur le plan des fondements théoriques que dans le milieu industriel. Récemment, plusieurs travaux ont proposé de mettre en œuvre les algèbres de processus dans des ADLs. En effet, les composants peuvent être vus comme des processus indépendants s'échangeant des messages. Ainsi, comme nous allons le voir dans la section 2.4, Wright et Darwin sont deux représentants de cette approche ; Wright utilise CSP pour spécifier le comportement des interactions au niveau des ports [GA94], tandis que Darwin est

un ADL qui base la sémantique de ses descriptions comportementales sur le π-calcul [MKG99a]. Ils ont en commun la possibilité d'accepter des outils qui vérifient la compatibilité de leurs descriptions des composants.

Les automates à états : ils jouent un rôle important dans la spécification et dans la vérification des systèmes discrets. Il s'agit des graphes orientés pour décrire et simuler le déroulement des systèmes. Les sommets de ces graphes sont appelés états et les arcs sont appelés transitions [dAH01].

Les automates ont été employés avec succès dans la pratique, notamment pour vérifier des protocoles de communication réseau. Plusieurs outils existent : par exemple MEC [Gri89] permet de calculer le produit synchronisé de deux automates et SPIN [Hol03] permet de vérifier la et de simuler des systèmes concurrents finis décrits en Promela [1](*Protocol Meta Language*).

L'approche par contrat peut être complétée par une modélisation du comportement des interfaces par des automates à états [Reu03]. Cela permet de définir des contraintes supplémentaires au niveau des interfaces des composants, par exemple, la définition de protocoles d'utilisation de ces derniers. Dans ce cas, le fournisseur d'un composant ne peut garantir le bon résultat de ses services que lorsque son client s'engage à respecter le protocole d'utilisation (c'est-à-dire la manière dont le composant doit être utilisé).

2.3.7.4 Niveau non-fonctionnel

Avec l'émergence des services dans le génie logiciel, plusieurs services peuvent résider au sein d'un même système, et être soumis aux exigences des clients. Afin de répondre à ces dernières, les services s'appuient sur l'utilisation de ressources, soit matérielles (bus de communication, mémoires, *etc.*), soit de ressources étant elles-mêmes d'autres services. Les clients (ou les services) sont dorénavant de véritables consommateurs de service.

Par ailleurs, le défi de la réutilisation des entités logicielles rend plus opaque l'identité des divers acteurs. Par conséquent, les acteurs réclament certaines garanties sans lesquelles un client peut être amené à dialoguer avec un service dont les ressources varient significativement. Inversement, un fournisseur peut voir son service, par exemple, subitement utilisé par un nombre important de clients et peut ne plus pouvoir faire face à leurs demandes. La notion de *contrat de QoS* (contrat de qualité de service), qui s'intéresse à la qualité de la relation entre un service et son client, devient donc un enjeu crucial. La QoS [ISO98] fait référence à de multiples propriétés telles que la performance, la disponibilité, la robustesse ou encore la sécurité.

Dans le monde des SOA et du CBSE, où les acteurs se connaissent peu, la garantie de ces propriétés est tout aussi importante que le traitement fonctionnel des services. Par exemple, si l'on prend un composant offrant un service qui distribue du contenu multimédia, la latence, la qualité du résultat ou la synchronisation entre l'image et le son sont des caractéristiques qui affectent le degré de satisfaction d'un

1. http ://spinroot.com/spin/Man/promela.html

utilisateur de ce service. Ces caractéristiques non-fonctionnelles sont généralement associées à la notion de qualité de service. Les caractéristiques de QoS doivent donc se faire rencontrer l'offre et la demande.

Toutefois, contrairement à certains standards établis au niveau structurel (tel que WSDL [CCMW01]), il n'existe pas de spécification ni de mécanisme officiellement reconnus par la communauté en ce qui concerne la spécification et le traitement de la QoS des services. Cependant, poussés par les enjeux liés à la QoS, plusieurs travaux ont été proposés permettant la réalisation et la mise en œuvre de contrats de QoS entre un client et un service [ISO98]. Ces travaux ont abouti à la spécification de divers modèles de contrat, de protocoles de gestion associés. Par exemple, WS-Agreement [LDK04] et WSLA [1] [KL03] basés sur SLA (présenté dans la section 2.3.5) permettent de spécifier un accord entre client et fournisseur de service portant sur le niveau de QoS que doit fournir ce dernier.

2.3.7.5 Synthèse

Comme nous pouvons le constater, il est nécessaire d'enrichir les interfaces des composants (souvent perçus comme des boîtes noires) pour permettre leur réutilisation. Ainsi, un langage de description d'interface de composants ne devrait pas se limiter à une description syntaxique des opérations fournies et requises par un composant. L'ajout d'autres informations sémantiques et quantitatives sur le comportement des composants vis-à-vis de leur environnement permet une meilleure compréhension du fonctionnement des composants et l'analyse de leur assemblage. Le tableau 2.3.1, largement inspiré de la classification de [BJPW99a], résume les différents niveaux de contrats.

Niveaux	Contrat	Propriétés	Langages
Niveau 1	Structurel	Compatibilité des signatures	IDL, WSDL, *etc.*
Niveau 2	Fonctionnel	Pré/post-conditions, invariants	Eiffel, iContract, UML/OCL, ConFract, *etc.*
Niveau 3	Comportemental	Synchronisation, communication	Wright, Darwin, SOFA, *etc.*
Niveau 4	Non-fonctionnel	QoS	WS-Agreement, WSLA, *etc.*

Tableau 2.3.1 – Différents niveaux de contrats [BJPW99a].

Certains langages, comme Eiffel, permettent une description des services au niveau fonctionnel, en intégrant dans leur syntaxe des pré, post-conditions et des invariants qui décrivent les services et les composants. Ces éléments de contrat sont utilisés défensivement à l'exécution pour lever des exceptions en cas de rupture de contrat et non pas préventivement par des outils de preuve ou de compilation sophistiqués. Sous réserve d'atomicité d'exécution des services, les contrats du niveau fonctionnel permettent de garantir un certain degré de confiance sur le bon fonctionnement des composants et services.

Toutefois, certaines implantations d'objet ou de composant incluent du parallélisme et les hypothèses d'atomicité ne sont pas toujours garanties. C'est le rôle du

1. Web Service Level Agreement

niveau comportemental des contrats de décrire les hypothèses de concurrence acceptées par les composants. Enfin, le niveau non-fonctionnel garantit des propriétés de QoS.

Comme toute classification, il s'agit essentiellement là d'un point de vue. La frontière est parfois floue entre deux niveaux. Par exemple, le type des paramètres des opérations considérés dans le premier niveau (structurel) des contrats pourrait, selon un point de vue tout à fait justifiable, apparaître dans le second niveau (fonctionnel). Par exemple, le langage B considère le typage des paramètres comme des prédicats dans la pré-condition de leur opération.

Nous rappelons que l'étude des contrats logiciels dans le cadre de cette thèse a été menée dans l'objectif de pouvoir vérifier les systèmes à base de composants tôt dans le cycle de développement. Les trois premiers niveaux de contrats semblent parfaitement compatibles avec notre objectif et seront largement exploités dans la suite (un focus supplémentaire est quand même réservé au niveau fonctionnel). En revanche, bien que les propriétés de QoS exprimées par le niveau non-fonctionnel soient également importantes dans les systèmes à base de composants, elles ne sont pas traitées ici. En effet, la QoS regroupe un nombre de facettes de l'application qu'il est difficile de borner (le temps d'exécution, la latence, *etc.*). En outre, les propriétés liées à la QoS d'un composant sont généralement difficiles à exprimer au niveau de l'interface d'un composant. Ces propriétés sont, dans la plupart du temps, fortement liées à des informations qui ne sont connues qu'au moment de l'exécution (*runtime*).

2.4 Mise en œuvre des contrats dans les modèles de composants

De nombreux modèles ont déjà été proposés pour le développement d'applications à base de composants [CL00]. Ces modèles se focalisent principalement sur les concepts-clés d'un modèle de composants tels que la définition de composants, leur assemblage, leurs interfaces *etc.* Outre l'étude des concepts, ces modèles représentent des expérimentations sur des fonctionnalités précises dans le but de mieux comprendre les composants et de contribuer par la suite à la définition de modèles industriels. L'existence d'un certain nombre de modèles de composants dits industriels ne justifie pas la maîtrise de tous les concepts sur les composants. En effet, ces modèles s'orientent en priorité vers la production, la diffusion et l'exécution des applications à base de composants, mais dans leur état actuel, ce ne sont que des extensions des modèles à objets et ils souffrent encore du manque de portabilité et d'une forte dépendance aux langages de programmation [OKS05b]. Par exemple, EJB [CFS03] et .NET [AS01b] sont deux représentants de cette catégorie. Ils sont largement utilisés dans des applications réelles de grande taille. Cependant, ils ne permettent pas de raisonner formellement au niveau composant ni au niveau des compositions de composants.

Dans notre étude, nous nous intéressons plus particulièrement aux modèles abstraits comme par exemple SOFA [BHP06], Fractal [BCL+06], UML 2.0 [Gro03] et BIP [GS05, Sif05] ou les ADLS tels que Wright [AG97], Darwin [MKG99a]. Nous

ne nous intéressons pas aux modèles spécifiques traitant du temps ou des propriétés non-fonctionnelles.

Dans la suite de cette section, nous exposons leur définition du composant et d'assemblage, leur façon de décrire les interfaces, leur capacité à modéliser explicitement les contrats et enfin, leurs plates-formes d'exécution et les outils qui leur sont associés.

2.4.1 .NET

Le modèle de composants .NET [1] proposé par Microsoft^{tm} vise le développement des applications largement réparties et hétérogènes (c'est-à-dire implantées en différents langages de programmation tels que Java, C#, VB, *etc.*). Ce modèle de composants est étroitement lié à la plate-forme *.NET Framework*.

Un composant .NET est appelé *assembly*[2]. D'un point de vue externe, un composant est caractérisé, entre autres, par un ensemble de ressources et de méthodes exportées. D'un point de vue interne, il est composé d'un ensemble de fichiers nécessaires à son exécution : il s'agit de fichiers d'implantation, de ressources et de description.

Une méthode d'intégration de contrats du niveau comportemental a été proposée dans [BS03], exprimée sous forme de programmes de modèles exécutables. Ces programmes sont des contrats permettant de vérifier la conformité d'une classe d'implantation avec sa spécification. La vérification des contrats, dite *runtime verification*, est dynamique. Elle contrôle l'exécution d'un composant dans le cadre de l'exécution d'un programme ayant des données spécifiques en entrée. Pour spécifier les programmes ainsi que leurs contrats et exprimer les propriétés sur l'interaction entre composants, le langage de spécification ASML (Abstract State Machine Language) est utilisé. ASML emploie des machines d'états abstraites qui se basent elles-mêmes sur des systèmes de transitions. ASML est également utilisé pour analyser et vérifier le comportement d'un composant écrit dans tout langage supporté par .NET. L'inconvénient de cette approche est que la vérification ne se fait qu'à l'exécution (*runtime*).

Une autre approche, ContractWizard [AS01b], exploite l'intégration d'Eiffel dans la plate-forme .NET. Elle permet d'appliquer des pré/post-conditions et des invariants sur les classes des langages à objets supportés par .NET, en produisant pour chacune d'entre elles un adaptateur en langage Eiffel. Cette approche présente l'avantage de ne pas nécessiter le code source des classes cibles. Ainsi, à partir d'un *assembly* (unité de déploiement .NET), il produit les *wrappers* Eiffel associés et les range dans un nouvel *assembly* qui prend alors la place du précédent.

2.4.2 EJB

EJB (*Enterprise Java Beans*) est spécifié par Sun Microsystems [CFS03]. Les EJB (appelés aussi *beans*) sont des objets Java dédiés à la construction d'applications

1. http ://www.microsoft.com/net
2. À ne pas confondre avec la notion d'assemblage vue dans la section 2.2.2

JEE [1]. Ils implantent les interfaces nécessaires pour interagir avec leur environnement d'exécution, le conteneur des EJB. Le code des EJB représente la partie métier d'une application JEE alors que le conteneur *container* offre des services de gestion, telles que la communication, les transactions et les aspects sécurité.

Un composant EJB se présente sous la forme d'un ensemble de classes Java regroupées dans un fichier de description. Chaque composant forme ainsi une unité réutilisable qui peut être combinée à d'autres pour former une application.

Une approche d'intégration de contrat dans les composants EJB a été proposée dans [VTS02] permettant de supporter les pré/post-conditions et les invariants. Les *containers* EJB [CFS03] sont modifiés pour inclure des intercepteurs dans leurs interfaces. La vérification des pré/post-conditions est alors déclenchée par les intercepteurs en amont et en aval de l'exécution des services des EJB. Les invariants sont exprimés au niveau de l'application et leur évaluation peut être effectuée régulièrement, ou déclenchée par des intercepteurs. A la différence de l'approche par contrat classique, l'évaluation des pré-conditions n'est pas réalisée du côté client, mais effectuée par le *container* EJB.

Dans [Reu03], les auteurs proposent d'enrichir les interfaces des composants par des automates (contrat du niveau comportemental) en profitant de la réflexivité des EJB. L'outil CoCoNut/J a ainsi été développé pour permettre l'adaptation du comportement des interfaces. Chaque interface est décrite par deux types d'automates :

- l'automate d'appel, qui définit un sous-ensemble de toutes les séquences d'appel valides sur l'interface,
- l'automate de fonction qui définit, pour chacune des fonctions de l'interface, toutes les traces possibles d'appel à des services extérieurs à son composant porteur.

Ainsi, quand un composant A s'adapte à un composant B, son automate d'appel restreint ses fonctionnalités à celles compatibles avec les automates de B. Le système peut adapter les fonctionnalités offertes d'un composant au moment de son insertion.

2.4.3 SOFA

SOFA (pour *SOFtware Appliance*) [BHP06] est un modèle de composants hiérarchiques où chaque composant présente classiquement un ensemble d'interfaces requises et fournies. La définition d'un composant est donnée par deux notions, *frame* et *architecture*. La *frame* est la vision en boîte noire du composant avec ses interfaces externes, l'*architecture* est une implantation spécifique d'un composant à l'aide de sous-composants. Une *frame* peut ainsi être implantée par plusieurs *architectures*.

Il existe quatre types de liaison entre les interfaces de composants :

1. le *binding* attachant une interface requise à une interface fournie entre deux composants de même niveau ;
2. le *delegating* lie une interface fournie d'un composant à une interface fournie d'un de ses sous-composants

1. Java Enterprise Edition, ou Java EE (anciennement J2EE)

3. le *subsuming* permet de lier une interface requise d'un sous-composant à une interface requise du composant l'englobant ;
4. le *exempting* exemptant une interface d'un sous-composant de n'importe quel lien. Notons que les trois premiers types de liens sont réalisés par des connecteurs.

Dans SOFA, des contrats du niveau comportemental (cf. section 2.3.7.3) ont été mis en œuvre par le biais des *behaviour protocols*. Ces derniers permettent de spécifier de manière formelle le comportement des composants ainsi que la communication entre eux. Le comportement d'un composant SOFA correspond à l'ensemble des enchaînements d'invocations de méthodes autorisés.

Les *behaviour protocols* reposent sur le principe selon lequel chaque appel ou retour de méthode forme un événement. Le *behavior protocol* d'une entité SOFA consiste en l'ensemble des traces d'événements qui peuvent être produites par l'entité. Un langage à base d'expressions rationnelles permet de former des expressions dénotant ces traces, à l'aide d'opérateurs pour désigner un appel de méthode, l'acceptation d'un appel de méthode, l'émission d'un retour ou l'acceptation d'un retour de méthode. Ces événements peuvent être associés à une instance d'interface et peuvent être combinés à l'aide d'un opérateur d'alternative, de séquence ou de fermeture transitive (boucle). Par exemple, une interface de manipulation d'un système de fichiers impose que l'ouverture d'un fichier précède toutes les autres actions, que la fermeture soit la dernière et qu'entre les deux s'exécute une suite quelconque de lecture et d'écriture : $(ouvrir\ ;\ (lire + ecrire)^*\ ;\ fermer)$.

Les *behaviour protocols* peuvent être définis au niveau des interfaces et au niveau des *frames* pour décrire la façon dont les composants doivent être utilisés :

- les protocoles d'interface (*interface protocol*) décrivent comment utiliser une interface.
- les protocoles de « composition » (*frame protocol*) décrivent comment assembler les différentes interfaces d'un composant et celles de ses composants internes.

Le compilateur SOFA génère alors l'*architecture protocol* à partir des *frame protocols* des sous-composants. L'intérêt de cette approche est double. Premièrement, un compilateur vérifie la cohérence des spécifications des différentes entités en interaction, par exemple en vérifiant qu'un *frame protocol* est conforme aux protocoles de toutes ses interfaces, et qu'un *architecture protocol* est conforme au protocole de la *frame* englobante. Deuxièmement, la vérification de la conformité des spécifications aux implantations se fait à l'exécution à l'aide d'automates simples.

Il est à noter que le formalisme des *behavior protocols* a été aussi récemment adapté au modèle Fractal [KAB+06] que nous présentons ci-après.

Le modèle SOFA intègre les contrats de niveau structurel et les contrats comportementaux basés sur les *behavioural protocols*. L'inconvénient est que la vérification de la conformité de l'implantation d'un composant par rapport à ces contrats ne se fait qu'en *runtime*. SOFA tente également d'ajouter la modélisation formelle des aspects non-fonctionnels, mais ils ne sont pas encore totalement intégrés dans le modèle.

2.4.4 Fractal

Le modèle de composants Fractal [BCL+06], développé dans le cadre du consortium *ObjectWeb* par France Telecom R&D et par l'INRIA, vise à autoriser la définition, la construction, la configuration et l'administration d'une architecture à base de composants.

Le modèle de composants Fractal possède deux niveaux d'abstraction. Le modèle général aborde de manière abstraite la notion de système à base de composants. Ce modèle comporte trois notions principales : la notion de *cell* (cellule), qui possède un *plasm* (contenu) et une *membrane*. Ce modèle est inspiré par une métaphore entre un composant et une cellule biologique. Le modèle concret spécialise ce modèle général pour obtenir un modèle proche des langages de programmation. Cette spécialisation introduit un système de type inspiré du langage de programmation ciblé.

Une extension du modèle Fractal, appelée **ConFract** [CRCR05], a été proposée pour intégrer des contrats du niveau fonctionnel. Un langage d'assertions exécutables, nommé CCL-J [1], a été utilisé dans le cadre de ConFract. Ce langage, partiellement inspiré d'OCL [OMG05], permet notamment d'exprimer des assertions avec une portée globale à un composant, éventuellement composite, et non pas uniquement sur une interface. L'utilisation de CCL-J permet également de séparer clairement les mécanismes basés sur les contrats et l'implantation des composants Fractal.

Dans ConFract, on distingue trois types de contrats :

1. un ***contrat d'interface*** est établi sur chaque connexion entre une interface requise et une interface fournie. Il regroupe les spécifications des deux interfaces, sachant que ces spécifications ne peuvent référencer que les méthodes de ces interfaces. Ce contrat s'apparente au contrat d'objet vu dans la section 2.3.4.
2. les ***contrats de composition externe*** sont posés sur la face externe de la membrane d'un composant. Ils sont formés des spécifications qui ne citent que des interfaces externes du composant. Ils expriment donc les règles d'utilisation et de comportement externe du composant.
3. les ***contrats de composition interne*** sont posés sur la face interne de la membrane d'un composant composite. Ils sont formés des spécifications qui ne citent que les interfaces internes du composant et les interfaces externes de ses sous-composants. Ils expriment les règles d'assemblage et de comportement interne de l'implantation.

Les différents contrats sont gérés par des contrôleurs de contrats, placés sur la membrane de chaque composant. Chaque contrôleur de contrats d'un composant composite prend en charge le cycle de vie et l'évaluation du contrat de composition interne du composite sur lequel il est placé, du contrat de composition externe de chacun des sous-composants, et du contrat d'interface de chaque connexion située dans son contenu.

L'implantation actuelle de ConFract repose sur l'implantation de référence de Fractal, *Julia* [BCL+04]. Le contrôleur de contrats utilise le mécanisme de *mixin*

1. Component Constraint Language for Java

fourni pour interagir avec les autres contrôleurs standards du modèle Fractal (*BindingController* et *ContentController* pour la construction et la mise à jour des contrats, *LifecycleController* pour la vérification d'invariants sur des composants). Des intercepteurs sont aussi placés sur les interfaces des composants afin de vérifier les pré/post-conditions.

2.4.5 UML 2.0

Après le succès de la première version d'UML (*Unified Modeling Language*), la deuxième version, notée UML 2.0[Gro03], introduit la notion de *diagramme d'architecture*, appelée aussi *diagramme de structure composite*. Avec l'ajout de ce diagramme, UML veut combler une des lacunes de la première version en permettant la spécification de l'architecture d'une application en terme de composants interconnectés. UML 2.0 définit deux types de composants : le composant basique (*Basic Component*) et le composant empaqueté (*Packaging Component*). Au niveau des caractéristiques, un composant basique étend la notion de classe. C'est une entité instanciable qui interagit avec son environnement par l'intermédiaire de ports ou d'interfaces. La description du comportement de ce composant est fondée sur des machines à états. La notion de composant empaqueté étend la notion de composant basique. Si le composant basique définit la manière dont le composant est vu à l'extérieur du composite, le composant empaqueté est une entité composée d'autres éléments liés de façon cohérente tout au long du processus de développement du logiciel.

Comme nous l'avons déjà expliqué dans la section 2.3.4, OCL est un langage de formalisation de contraintes (initialement proposé dans le contexte d'UML) qui se veut accessible aux non spécialistes des techniques formelles. La version 2.0 d'UML ne modifie pas fondamentalement la philosophie du langage OCL, mais propose un certain nombre d'améliorations [QT06]. Une clause *body* permet de décrire des requêtes spécifiant la valeur à retourner. Ceci permet d'utiliser OCL non plus seulement comme un langage de contraintes, mais comme un langage de programmation interprétable sur des modèles UML. Dans OCL 2.0 [OMG05], il est aussi possible de contraindre l'évolution de la valeur d'attributs. D'autre part, les structures de données ensemblistes ne sont plus aplaties lors de leur parcours, ce qui permet de disposer de listes. Des quantificateurs universels ou existentiels portant sur les extensions de classes sont également disponibles.

UML 2.0 est un modèle abstrait qui ne possède pas de plate-forme propre, mais propose des mécanismes de génération de codes vers des plates-formes à composants existantes.

2.4.6 Wright

Wright [All97] a été l'une des premières approches qui permettent la description des contrats comportementaux de composants. Un composant en Wright est une unité abstraite localisée et indépendante. La description d'un composant contient deux parties importantes qui sont la partie externe (*interface*) et la partie calcul (*computation*). L'interface consiste en un ensemble de ports, chacun représentant une

interaction avec l'extérieur à laquelle le composant peut participer. La spécification d'un port décrit deux aspects de l'interface du composant :

- elle décrit partiellement le comportement attendu du composant.
- elle décrit ce que le composant attend du système dans lequel il va interagir.

Le calcul décrit ce que le composant fait du point de vue comportemental. Il mène à bien les interactions décrites par les ports et montre comment elles se combinent pour former un tout cohérent. C'est sur cette spécification que l'analyse des propriétés du composant va être basée.

La *configuration* permet de décrire l'architecture d'un système en regroupant des instances de composants et des instances de *connecteurs*. Wright supporte la composition hiérarchique. Ainsi, un composant peut être composé d'un ensemble de composants.

La mise en œuvre des contrats comportementaux dans Wright est décrite à l'aide du langage CSP (Communicating Sequence Process) [Hoa83]. Ce langage a été créé en 1978 par Hoare dans un souci de formalisation théorique (cf. section 2.3.7.3). Il est basé sur une logique de composition des actions (logique de Hoare) [Hoa69]. La description du comportement dans Wright permet une analyse de l'assemblage de composants. Cette analyse concerne deux types de propriétés au niveau de l'architecture :

- les propriétés de sûreté : elle s'assurent que l'assemblage ne produit pas d'interblocage. Elles permettent également de vérifier des propriétés, définies par l'utilisateur, sur l'ordonnancement des messages ;
- les propriétés de vivacité : elle garantissent que le système évolue (afin d'effectuer ses fonctionnalités).

La correction d'un système est donc examinée principalement via ces propriétés. Les divers outils de vérification formelle supportant CSP (par exemple FDR [FW07]) peuvent être réutilisés avec profit pour l'analyse formelle d'architectures logicielles décrites en Wright. L'expressivité du modèle Wright est donc limitée à celle de CSP.

Du point de vue des vérifications, Wright autorise la vérification d'absence d'interblocage ainsi que d'autres vérifications de cohérence. Il permet notamment de vérifier la compatibilité des parties connectées sur la base de la compatibilité de leurs protocoles d'échange de messages. Ces vérifications sont réalisées à l'aide d'un outil de *model-checking*, pour lequel les outils de Wright (Wr2FDR par exemple) produisent les entrées. L'ensemble des vérifications se fait lors de la phase de conception de l'architecture.

Cependant il n'y a pas de générateur de code, ni de plate-forme permettant de simuler l'application. En effet, Wright ne s'intéresse ni au déploiement ni à l'exécution d'une architecture logicielle à base de composants. Il se concentre plutôt sur la description des modules logiciels et leurs interactions, son objectif étant orienté vers la vérification formelle d'une architecture logicielle.

2.4.7 Darwin

Darwin [MKG99a] est un ADL qui base la sémantique de ses descriptions structurelles sur le π-calcul. Dans Darwin, chaque composant est considéré comme une

entité atomique d'exécution. La composition hiérarchique de composants est présentée comme la mise en parallèle des processus qu'ils représentent.

L'interaction entre les composants se fait par l'intermédiaire des services offerts et requis par le composant. Le concept de connecteur n'est pas explicitement présent dans Darwin. En effet, chaque interaction est représentée par un lien entre un service fourni et un service requis de composants différents. Les services n'ont aucune connotation fonctionnelle, ils désignent seulement le type d'objet de communication utilisé et autorisé à utiliser une fonction du composant. Ces types d'objets sont définis par le support d'exécution réparti appelé Regis et sont limités. Parmi les types d'objets, le port est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre composants répartis ou non. Dans une description d'architecture exprimée avec Darwin, la section « port » caractérise le type d'objet de communication, et la section « signature » décrit la signature de la fonction du composant. Darwin effectue par défaut un simple test d'équivalence des noms et des types avant la connexion, vérifiant que le service requis est compatible avec le service fourni.

Afin d'offrir une hiérarchie au sein d'une application, Darwin contient deux sortes de composant :

- le composant primitif, qui est avant tout une entité d'encapsulation de fonctions et de données d'un module logiciel. Il est défini par son nom et son interface qui déclare les services requis et fournis par le composant métier,
- le composant composite, qui contient des composants primitifs et des composants composites. C'est donc une unité de configuration. Il décrit les interconnexions entre les composants. Ainsi, la structure finale d'une application est représentée par un composant composite.

Plus généralement, une architecture Darwin se traduit en un programme de π-calcul dont la validité est synonyme de celle de l'architecture. Ainsi Darwin base sa description architecturale sur l'échange de messages entre processus. Le comportement des composants est spécifié à l'aide d'une autre algèbre de processus nommée FSP (Finite State Processes) [Mag99] et de diagrammes de transitions d'état exprimés en LTS (Labelled Transition System) et associés aux messages des FSP. Le diagramme intervient comme un outil de visualisation des comportements globaux qui, d'après l'expérience des auteurs, occupe un rôle important dans la vérification et la recherche d'erreurs dans la conception. Des outils associés ont été développés pour la vérification de propriétés de sûreté et de vivacité lorsque des modèles sont composés.

L'association de la définition des services, de la dynamique de l'architecture et du comportement des composants fait de Darwin un modèle d'architecture précurseur, très intéressant pour la spécification et l'analyse d'applications distribuées.

La présence des contrats du niveau comportemental dans Darwin permet comme dans Wright de détecter la présence d'interblocage et de vérifier les propriétés de sûreté ou de vivacité. Au niveau des propriétés de vivacité, Darwin définit la notion de propriété de progression. Cette propriété permet dans le cas où l'on donne une priorité d'ordonnancement des processus de garantir que pour une infinité d'exécutions d'un processus, on est sûr qu'au moins une action définie au niveau de la propriété

va s'exécuter une infinité de fois. Ce type de propriété qui est un sous-ensemble des propriétés de vivacité permet de mettre en évidence, dans les cas où la priorité des composants est définie, le fait que certaines actions n'ont jamais l'occasion de s'exécuter.

2.4.8 BIP

BIP [Sif05] est un modèle de composants dont l'objectif est de construire des systèmes corrects par composition de composants hétérogènes.

BIP adopte une méthodologie à trois couches, *Behaviour, Interaction,* et *Priority* (d'où le nom BIP), pour la construction des composants :

1. une couche basse *Behaviour* dans laquelle on décrit des comportements sous la forme d'un système de transitions étiquetées. Chaque transition est étiquetée par un port, une garde et une fonction. La fonction, qui est exécutée lorsque la transition est tirée, permet de modifier l'état local du composant et est exprimée en langage C. Un composant possède un ou plusieurs ports qui représentent les seuls points d'interaction possibles. Les composants peuvent être atomiques (*atom*) ou composites (*compound*) ;
2. une couche intermédiaire *Interaction* contenant des connecteurs qui relient des ports de différents composants entre eux, et définit un ensemble d'interactions possibles entre les comportements du niveau précédent ;
3. une couche haute *Priority*, consacrée à la description d'un ensemble de règles de priorités pour l'ordonnancement des interactions. Elle permet donc de réduire l'ensemble des interactions possibles.

En effet, les ports sont des exportations des actions internes. Par défaut, les ports sont implicitement internes.

La composition de composants BIP est binaire et consiste en la composition deux à deux des trois niveaux (*behaviour, interaction,* et *priority*) des différents composants. Le résultat est aussi un composant à trois niveaux. La composition est ainsi incrémentale.

BIP est accompagné d'une plate-forme qui permet de générer du code (C++) exécutable et la vérification formelle de propriétés sur un graphe d'états engendré à partir de la description des composants.

Les modèles BIP peuvent être analysés en utilisant la plate-forme IF [BGM02] qui offre de puissantes techniques basées sur l'exploration d'états (*model-checking*). Les modèles internes générés pour les composants BIP sont traduits en format IF. Les propriétés analysées sur des composants BIP sont : *deadlock-freedom*, *state invariants* et *schedulability*.

2.5 Synthèse comparative

Dans cette section, nous dressons un bilan récapitulatif des différents modèles de composants abordés dans ce chapitre, suivant un ensemble de critères que nous

jugeons pertinents pour notre problématique, à savoir la vérification des systèmes à base de composants.

2.5.1 Entités exhibées par le composant

Comme nous l'avons déjà vu dans la section 2.2.1.2, « a component is a static abstraction with plugs » [ND95]. Selon le modèle de composants considéré, les fonctionnalités d'un composant sont exposées à travers différentes entités (*plugs*) comme : les ports, les interfaces, les interfaces de port [ACN02a], les protocoles [PV02], les services [MKG99a], les contrats [BJPW99a]. Le tableau 2.5.2 présente une comparaison des points d'accès d'un composant dans les différents modèles que nous avons présentés.

	.NET	**EJB**	**SOFA**	**Fractal**	**UML2.0**	**Wright**	**Darwin**	**BIP**
Entité exhibée	opérations fournies	opérations métiers, événement	interfaces (protocole)	interfaces	port/ interface	port	service	port
Distinction offert/requis	non	non	oui	oui	oui	oui	oui	non
Formalisme	IDL	Java	CDL[a]	Fractal IDL	UML	Wright	Darwin	BIP

[a] Component Definition Language

Tableau 2.5.2 – Description des points d'accès de composants.

La notion d'interface est utilisée dans EJB (interface Java qui définit les opérations-métiers), dans les composants .NET et Fractal (en utilisant IDL). Cependant, contrairement à EJB et .NET, Fractal permet d'expliciter la notion d'interface requise. Il définit également des interfaces de contrôle. Les interfaces dans SOFA sont accompagnées de la notion de protocole permettant de préciser leur comportement. En UML, un composant fournit ou requiert des interfaces à travers ses ports mais il peut aussi le faire directement (sans passer par un port) [MP06].

Wright et BIP utilisent des ports d'interaction dotés d'une description formelle définissant leur comportement. Enfin, le concepte du service est utilisé dans Darwin et dans [DCL08, Krä08, EFLA04].

En comparant ces différents modèles, nous notons que certains modèles (par exemple, SOFA, Fractal et UML 2.0) supportent l'implantation de types d'interfaces, et déclarent explicitement leurs interfaces requises, alors que d'autres n'offrent que des interfaces de type appel de méthode (.NET et EJB). De plus, dans la plupart des modèles de composants, les interfaces sont décrites par une interface Java. Les interfaces de composants sont donc souvent, par abus de langage, confondues avec les interfaces Java alors qu'elles remplissent des fonctions différentes (cf. section 2.2.1.2).

2.5.2 Description interne du composant

La plupart des approches à base de composants ne couvrent pas l'ensemble du cycle de développement d'une application, depuis sa spécification, considérée comme

le plus haut niveau d'abstraction, jusqu'à son code binaire, représentant le plus bas niveau. Notons que cette différence d'abstraction est due aux objectifs spécifiques des différentes approches (cf. tableau 2.5.3).

	Notation	Formalisme	Implantation	Plate-forme	Domaine d'application
.NET	code source	Java, C#, VB, *etc.*	Java, C#, VB, *etc.*	.Net Framework	développement des applications largement réparties et hétérogènes (Java, C#, VB, *etc.*)
EJB	code source	Java	Java	JEE	développement des applications distribuées (JEE)
SOFA	abstraite	CDL + *behaviour protocole*	-	DCUP, SofaNode	reconfiguration dynamique des applications à composants
Fractal	abstraite	Fractal-ADL	Java, C/C++, .Net lang.	Julia, Aokell, Think, FractNet	conception et distribution des applications à composants
UML2.0	abstraite, graphique	UML (statecharts)	-	-	modélisation des applications à composants
Wright	formelle	CSP	-	-	analyse de protocoles de communication
Darwin	abstraite, formelle	FSP	C	Regis	applications réparties
BIP	abstraite, formelle	LTS, RdP	C++	IF	construction des systèmes corrects par composition

Tableau 2.5.3 – Description interne du composant

Le modèle EJB définit un composant comme un objet Java qui fournit un ensemble d'opérations-métiers et un certain nombre d'opérations de contrôle. De manière similaire à un composant EJB, un composant .NET est un objet implanté par n'importe quel langage de programmation supporté par la plate-forme .NET (Java, c#, VB, *etc.*). Les EJB et .NET sont donc des approches fortement liées à l'environnement.

En revanche, les modèles de composants Fractal et SOFA sont indépendants de l'implantation, mais proposent tout de même des plates-formes pour leurs composants. Ainsi, SOFA est accompagné de sa plate-forme DCUP et Fractal permet de choisir sa plate-forme cible en fonction du contexte du projet (Julia, Think, FractNet, *etc.*).

Comme beaucoup d'ADLs, Wright et Darwin sont des approches qui s'intéressent principalement à la spécification des architectures en utilisant des langages suffisamment abstraits et indépendants de toute plate-forme d'exécution (CSP, π-calcul, FSP, LTS). En revanche, contrairement à Wright, Darwin propose toutefois une plate-forme (Regis) pour ses composants.

Grâce à son caractère abstrait et sa notation graphique, UML s'inscrit dans la même lignée que les ADLs mais ne propose pas de support formel, contrairement à de nombreux ADLs. Il ne propose pas non plus de plate-forme d'exécution propre. BIP, quant à lui, est assisté par une plate-forme IF qui permet de générer du code (C++) et d'effectuer la vérification formelle basée sur le *model checking*.

2.5.3 Support de compositions

La composition est traitée de diverses façons selon les caractéristiques du modèle considéré. Nous avons identifié deux éléments-clés sur lesquels se base la composition :

- les points d'accès des composants : ils peuvent être, comme nous l'avons déjà vu, des ports (Wright, BIP), des interfaces (Sofa, Fractal) ou des services (Darwin). Il peut même y avoir plusieurs niveaux comme dans UML 2.0 où un port peut proposer plusieurs interfaces ;
- les entités établissant la liaison : elle peuvent être de simples liens (Fractal, Darwin) ou des connecteurs (Wright, Sofa, BIP).

La composition dans EJB (comme dans .NET) est impérative à travers le langage de programmation supporté. L'architecture d'une application EJB (ou .NET) est donc noyée dans le code d'implantation. Cette limitation vient du fait que les modèles EJB ou .NET n'expriment de façon explicite ni les interfaces requises, ni les liaisons entre composants. Les interactions sont simplement des appels de méthode. De plus, les modèles EJB et .NET ne permettent pas la hiérarchisation des composants, ce qui pose problème au niveau de l'analyse des systèmes complexes.

	.NET	EJB	SOFA	Fractal	UML2.0	Wright	Darwin	BIP
Entités connectées	opérations	opérations métiers	interfaces	interfaces	port/interfaces	ports	services	ports
Type de liaisons	appel de méthode	appel de méthode	-connecteur (*binding, delegation, subsuming*), *-exempting*	*binding*	-connecteur d'assemblage, - connecteur de délégation	connecteur	invocation de service	connecteur
Hiérarchique	non	non	oui	oui	oui	oui[a]	oui	oui
Langages	Java	Java	CDL	Fractal-ADL	UML	ADL (Wright)	ADL (Darwin)	BIP
Communication	synch.	synch., asynch.	synch.	synch., asynch	synch., asynch.	synch., asynch.	asynch.	synch, asynch.

[a] se fait par raffinement

Tableau 2.5.4 – Support de composition.

En revanche, la composition peut être réalisée de façon déclarative à travers des langages dédiés comme dans SOFA, Fractal ou la plupart des ADLs. Les composants SOFA peuvent être composés verticalement, par encapsulation ; l'approche de structuration est descendante, c'est la méthode de conception primordiale préconisée dans SOFA. La composition horizontale dans SOFA consiste en la composition parallèle des comportements (*protocols*) des composants concernés ; ils communiquent alors par échanges synchrones de messages.

Fractal se veut un modèle hiérarchique et récursif de composants supportant le partage de sous-composants. La composition horizontale se fait par une liaison (*binding*) entre une interface requise et une interface offerte ; dans ce cas, les composants Fractal peuvent communiquer par appel de méthode.

Dans UML 2.0, la composition se fait, comme dans SOFA et Fractal, de façon horizontale et verticale. La première se fait par des connecteurs d'assemblage, et la seconde par des connecteurs de délégation.

Dans Wright, la composition horizontale est définie par des composants reliés sur leurs ports par des connecteurs. La composition verticale, quant à elle, est déléguée

au raffinement.

Darwin permet de décrire l'architecture de systèmes par des hiérarchies de configurations qui représentent des ensembles d'instances de composants connectées entre elles. Les connexions se faisant à travers de simples liaisons entre services requis et fournis, il ne propose donc pas le concept de connecteur explicite.

Dans BIP, la composition horizontale consiste à connecter les composants deux à deux grâce à une *glue*, le résultat est aussi un nouveau composant. La composition verticale est alors hiérarchique, enrichie par des règles spécifiques de calcul du modèle d'interaction (exprimées dans la *glue*) pour le composant résultant.

Il est intéressant de noter que les entités mises en relation de délégation restent, dans le cas général, inchangées (appelées délégation pure dans certains modèles). Néanmoins, dans la pratique, les entités des composants de base doivent être adaptés à l'environnement-cible de leur composite (qui n'est pas forcément celui pour lequel elles ont été conçues).

2.5.4 Support de contrats, de vérification et d'outils

Comme nous l'avons déjà vu en détail dans la section 2.4, les contrats ont fait l'objet de plusieurs travaux dans les modèles de composants. Ces derniers intègrent les contrats en utilisant des formalismes, des sémantiques et des mécanismes parfois différents.

La meilleure solution serait d'avoir un modèle de composants commun qui prendrait en compte les quatre niveaux proposés par Beugnard et al. [BJPW99a] dans un cadre cohérent. Mais, à notre connaissance, aucun modèle de composants existant n'a pu atteindre cet objectif.

A titre d'illustration, le tableau 2.5.5 montre comment ces niveaux de contrats sont intégrés dans les modèles de composants que nous avons étudiés. Il montre également les propriétés exprimées par les différents contrats intégrés, les formalismes et les techniques utilisés ainsi que les outils associés.

La plupart des modèles de composants "industriels" existants, tels que EJB ou .NET, sont basés uniquement sur le premier niveau qui assure un typage correct des interfaces fournies. Cependant, ces modèles proposent quelques services non-fonctionnels tels que la gestion de la persistance, la sécurité, les transactions, *etc.*, mais la liste de ces derniers est figée et non extensible. En effet, il est impossible d'ajouter de nouveaux services pour couvrir n'importe quelle propriété non-fonctionnelle. Même si des extensions ont été proposées pour prendre en compte les contrats fonctionnels dans de tels modèles, ces extensions ne vérifient le respect des contrats qu'en *runtime*. Les contrats comportementaux sont difficiles à intégrer dans de tels modèles car le comportement des composants est décrit directement par des langages de programmation. En effet, les contrats comportementaux intégrés dans l'approche CoCoNut/J sous forme d'automates s'intéressent plus à la problématique de l'adaptation qu'à celle de la vérification.

Le modèle SOFA intègre les contrats de niveau structurel et les contrats comportementaux basés sur les *behavioural protocols*. L'inconvénient est que la vérification de la conformité de l'implantation d'un composant par rapport à ces contrats ne

	Propriétés	Formalismes	Contrats				Techniques	Phase	Outils
			struct.	fonct.	comport.	QoS			
.NET	conformité d'une classe d'implantation avec sa spécification	machines d'états abstraites (AsmL)	✓	✗[a]	✗	✓	simulation, test	*runtime*	SpecExplorer, Contract-Wizzard
EJB	persistance	-	✓	✗	✗[b]	✓	-	*runtime*	JEE
SOFA	conformité des protocoles	*behavior protocol*	✓	✗	✓	✗	inclusion d'ensembles (de traces)	*runtime*	DCUP
Fractal	conformité des contrats fonctionnels	ConFract (CCL-J)	✓	✓[c]	✗	✗	sous-typage	*runtime*	ConFract
UML 2.0	-	*statecharts*	✓	✓[d]	✓[e]	✗	-	-	-
Wright	la cohérence entre le comportement d'un composant et celui de ses ports, l'absence d'interblocages	CSP	✓	✗	✓	✗	*model checking*, simulation	*design time*	FDR (de CSP)
Darwin	vivacité et sûreté	FSP, automates de Büchi	✓	✗	✓	✗	*model checking*	*design time*	SAA [1], LTSA [2]
BIP	-protocole, -détection d'interblocage	LTS, RdP	✓	✓	✓	✗	simulation	*design time*	IF-toolset

[a] une extension basée sur Eiffel a été proposée par ContractWizard

[b] une extension basée sur les automates a été proposée dans CoCoNut/J

[c] en utilisant CCL-J dans l'extension ConFract

[d] les contraintes OCL peuvent être utilisées comme des contrats fonctionnels

[e] les statecharts et les diagrammes d'activités peuvent être vus comme des contrats comportementaux

Tableau 2.5.5 – Support de contrats.

se fait qu'en *runtime*. SOFA tente également d'ajouter la modélisation formelle des aspects non-fonctionnels, mais ils ne sont pas encore totalement intégrés dans le modèle.

Le modèle ConFract [CRCR05] est une extension du modèle de composants Fractal qui se concentre principalement sur l'annotation des interfaces avec des pré et des post-conditions. Ces contrats ne sont pas intégrés dans le modèle Fractal, mais sont des entités indépendantes qui utilisent un langage d'assertions exécutables CCL-J et ne sont vérifiés qu'au moment de l'exécution.

Les modèles Wright et Darwin, comme de nombreux autres ADLs, supportent des notations formelles pour décrire l'interaction entre les composants. Même si cela permet de couvrir les contrats comportementaux, aucune étude n'a, à notre connaissance, abordé le problème des contrats fonctionnels sous forme de pré/post-conditions dans ces approches. BIP décrit, en plus des contrats comportementaux, des contrats fonctionnels exprimés sous forme de pré/post-conditions mais ne propose aucune vérification ni d'outils basés sur ces derniers.

Il existe également certains modèles [Tou05] qui prennent en compte la qualité de service, mais ne traitent pas forcément les contrats fonctionnels ni les contrats comportementaux.

Dans le tableau 2.5.5, on peut facilement constater que (*i*) la notion de contrat est maintenant largement répandue dans les modèles de composants. En effet, tous les modèles intègrent, par nécessité, les contrats structurels pour assurer le typage correct des interfaces offertes et requises (même nom de service, même nombre d'arguments, mêmes types, *etc.*) ; (*ii*) les contrats de niveaux supérieurs sont encore peu

exploités de façon intégrée ; et finalement, (*iii*) la plupart des approches intégrant les contrats fonctionnels, soit ne proposent pas d'outils pour les vérifier (comme UML 2.0), soit se limitent à la vérification en *runtime*.

Un des objectifs de cette thèse est de réunir plusieurs niveaux de contrats (structurels, fonctionnels et comportementaux) dans un même modèle de composants, et de proposer des techniques ainsi que des outils de vérification des propriétés avant même l'implantation des composants et leur composition. Nous ne traitons donc pas des contraintes telles que la qualité de service, car elles dépendent souvent de certaines informations qui ne sont disponibles qu'au moment de l'exécution.

2.6 Conclusion

L'étude menée dans ce chapitre nous a permis de constater que l'approche à composants est bâtie sur un cycle de développement qui fait une séparation claire entre les activités de développement de composants et celles de leur assemblage. Ces deux activités sont réalisées par des acteurs spécialisés qui peuvent ne pas être les mêmes. L'existence de ces deux activités explique diverses caractéristiques des composants, notamment l'existence d'une vue externe qui donne la vision d'un composant en tant que boîte noire. C'est pourquoi le composant doit avoir des points d'accès clairement définis. En outre, afin de faciliter son utilisation (et sa réutilisation), il doit être décrit (spécifié, annoté, *etc.*) précisément pour permettre la compréhension de son fonctionnement sans avoir à accéder à son implantation interne. Pour répondre à ce besoin, nous avons étudié l'approche basée sur la notion de contrat dans le développement des logiciels.

Nous avons également présenté différentes propositions de modèles de composants représentatifs, dans l'objectif de ressortir les propriétés communes à prendre en considération lors de l'analyse d'un système à base de composants. Nous citons notamment les types d'interfaces exposées par les composants et les contrats associés (afin d'exprimer les propriétés à vérifier), le niveau d'abstraction et la composition de composants (afin de simplifier la vérification), le support de vérification et les outils associés (afin d'automatiser la vérification).

En comparant les différents modèles existants, nous avons identifié trois principaux obstacles à la vérification des systèmes à base de composants :

- premièrement, il n'existe pas de modèle standard de composants. Ceci est principalement dû au fait que l'ensemble des acteurs concernés ont des priorités différentes concernant les caractéristiques devant être offertes par de tels modèles ;
- deuxièmement, la plupart des travaux autour de la vérification dans les modèles existants ne s'intéressent qu'à une partie de la problématique de vérification (aspects structurels, aspects fonctionnels, ou aspects comportementaux) ;
- troisièmement, bien qu'un certain nombre d'approches aient été proposées pour vérifier les systèmes à base de composants, cette vérification n'intervient malheureusement qu'après l'étape d'implantation [1] (*runtime verification*).

1. Ce n'est pas le cas dans certains ADLs (par exemple Wright et Darwin).

Pour pallier ces obstacles, nous proposons (*i*) de nous baser sur un modèle simple (ne supportant que les concepts essentiels et réellement exploités dans notre problématique), suffisamment abstrait (afin de maîtriser la complexité des systèmes), extensible (pour pouvoir intégrer les différents niveaux de contrats), formel (permettant de raisonner sur des propriétés exprimées au préalable et éventuellement automatiser leur vérification) et pouvant être raffiné (pour tirer profit des plates-formes existantes) ; (*ii*) une démarche de vérification *a priori* (afin de réduire, contrairement aux vérifications en *runtime*, le coût de la preuve et du test) des systèmes à base de composants basée sur les contrats qui prend en compte non seulement les aspects structurels mais également les aspects fonctionnels et comportementaux ; (*iii*) une solution pour la vérification *a priori* des contrats fonctionnels. Contrairement à la plupart des approches existantes, notre objectif est d'intégrer tous ces aspects dans un même modèle de composant et non pas sous forme d'extensions comme ConFract (pour Fractal) ou ContractWizzard (pour le .NET) ou OCL (pour UML 2.0) *etc.*

Ainsi, dans le chapitre suivant, nous introduisons le modèle Kmelia sur lequel nous nous sommes basés dans cette thèse. Ensuite, nous présentons dans le chapitre 4 l'enrichissement que nous avons apporté à ce modèle en terme de support de contrats (proposition *i*). Le chapitre 5 présente notre approche globale de vérification (proposition *ii*) et enfin les chapitres 6 et 7 montrent la manière dont les propriétés exprimées par les contrats fonctionnels sont vérifiées ainsi que les outils associés (proposition *iii*).

Chapitre 3

Kmelia : un modèle de composants multi-services

L'objet de ce chapitre est de décrire le modèle de composants Kmelia sur lequel se basent les travaux présentés dans cette thèse. Nous commencerons par introduire le modèle de façon informelle. Ensuite nous détaillerons les concepts de base. Nous établissons également une synthèse des propriétés vérifiées ainsi que les outils associés. L'ensemble sera illustré à travers un cas d'étude inspiré du CoCoME (Common Component Modelling Example).

3.1 Présentation informelle du modèle Kmelia

Kmelia est un modèle et un langage à composants multi-services [AAA11]. Les services décrivent des fonctionnalités avec la possibilité d'interagir avec d'autres services. Le modèle Kmelia partage des caractéristiques communes avec les approches à composants [AG97, MT00] : les composants, les services et les assemblages. Mais Kmelia se distingue d'autres approches par certaines caractéristiques spécifiques. Les composants Kmelia sont abstraits, indépendants de leur environnement et par conséquent non directement exécutables. Kmelia permet de modéliser des composants logiciels et des architectures logicielles (assemblages) avec leurs propriétés. La hiérarchisation des services et des composants est basée sur l'encapsulation et permet une bonne lisibilité, une flexibilité et une traçabilité dans la conception des architectures [AAA06a]. Kmelia peut également servir de support pour l'étude de propriétés (interopérabilité, composabilité) de modèles de composants et de services. Les modèles peuvent, par la suite, être raffinés vers des plates-formes d'exécution.

Les travaux autour de Kmelia ont débuté en 2005 et ont donné lieu à un modèle de base [AAA06c] comparable à un noyau, avec très peu de concepts (composants, service, assemblage). Ce modèle de base a été successivement enrichi ; d'abord par une couche protocole [AAA07b] permettant de définir des enchaînements licites de services. Cette extension repose sur la spécialisation de la notion de service. Ensuite, il a été ajouté une extension aux services partagés (induisant des canaux multi-points) et à la communication multiple. Cette extension est proposée dans [AAA08].

Dans le cadre de cette thèse, nous avons enrichi le modèle Kmelia en lui permet-

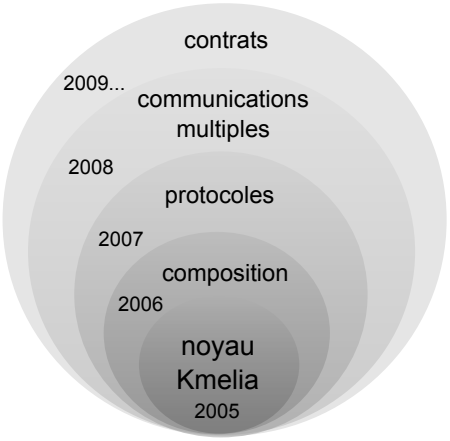


FIGURE 3.1 – Enrichissement du modèle Kmelia.

tant d'intégrer les contrats de façon explicite [AAM09]. Ainsi, nous avons étendu la grammaire Kmelia pour prendre en compte la définition des types, des expressions, et des assertions. Bien qu'elle soit entremêlée tant dans le modèle Kmelia que dans la plate-forme COSTO, nous avons toutefois séparé notre contribution en terme de présentation (cf. chapitres suivants). En revanche, pour des soucis de clarté et de cohérence, les définitions présentées dans ce chapitre sont basées sur l'état actuel du modèle Kmelia (y compris nos contributions).

3.1.1 Exemple de spécification en Kmelia

Dans la suite de ce chapitre, nous allons nous baser sur un cas d'étude inspiré du CoCoME [1] (Common Component Modelling Example) afin d'illustrer les concepts de base du modèle Kmelia que nous aurons présentés. L'application CoCoME est un système de vente en caisse avec les dispositifs usuels (caisse, scanner, imprimante, lecteur de carte...). Le processus de vente inclut des fonctionnalités directement liées aux achats du client (la lecture optique des codes de produits, le paiement par carte ou en espèces, *etc.*) et d'autres tâches telles que la gestion du stock et la génération des rapports. L'énoncé original du cas comprend des diagrammes UML (cas d'utilisation, composants, classes).

Nous retenons de cette étude de cas un extrait de la modélisation qui illustre les principaux concepts de Kmelia [AAA11]. Un premier assemblage est mis en évidence dans la figure 3.2. Il comprend le composant applicatif principal (*CashDeskApplication* relié à des dispositifs d'entrée/sortie, un composant de lecteur de carte *CardReader* et un composant de lecture optique *Scanner*. Ces composants décrivent dans leur interface des services offerts (rectangles grisés) et des services requis (rectangles blancs). Des liens d'assemblage, représentés graphiquement par des traits, relient par paire un service offert à un service requis. Par exemple le service offert `scan` est lié au service requis `scan_card`. Les flèches à l'intérieur des composants représentent les appels de service pour le service *process_sale*. Le composant *CashDeskApplication* offre un service de vente *process_sale*. Ce dernier fait appel à d'autres services tels

1. Ce cas d'étude a été utilisé pour comparer des modèles à composants [RRMP08]

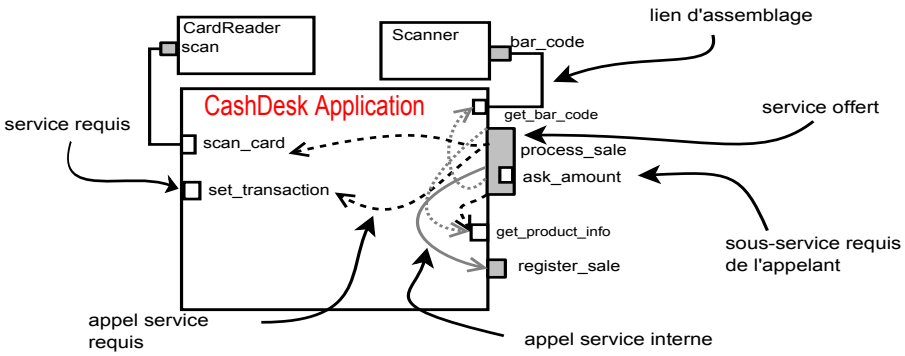


FIGURE 3.2 – Description Kmelia d'un assemblage partiel du cas CoCoME.

que *ask_amount()* requis de son service appelant, ou bien *set_transaction*, *scan_card* requis d'autres composants. Nous reviendrons sur le détail de l'exemple au fur et à mesure que nous introduirons les différents concepts du modèle Kmelia.

3.2 Préliminaires

Dans la suite de ce chapitre, nous allons utiliser des notations mathématiques inspirées de Z [Spi89]. Soient X, Y, X_1 et X_2 des ensembles. $X \leftrightarrow Y$ dénote l'ensemble des relations de X vers Y ($x \mapsto y$ dénote le couple (x, y), un élément de la relation) ; $X_1 \uplus X_2$ dénote l'union disjointe de deux ensembles X_1 et X_2 (c'est-à-dire l'union de X_1 et X_2 sachant que $X_1 \cap X_2 = \emptyset$) ; $X \nrightarrow Y$ est l'ensemble des fonctions partielles de X dans Y ; $X \rightarrow Y$ est l'ensemble des fonctions totales de X dans Y ; $X \rightarrowtail\!\!\!\!\!\!\rightarrow Y$ est l'ensemble des bijections partielles de X dans Y ; id est la relation d'identité. Si r est une relation ($r : X \leftrightarrow Y$), dom(r) et ran(r) représentent respectivement le domaine et le co-domaine de la relation r ; $r(\!|A|\!)$ est l'image relationnelle de l'ensemble $A \subseteq X$; $E \lhd r$ et $r \rhd F$ représentent la restriction respectivement du domaine et du co-domaine de la relation r avec $E \subseteq X$ et $F \subseteq Y$.

Définition 3.1 (Espace d'états) *Un espace d'états $\mathcal{W}$ est défini par un ensemble de constantes et de variables typées, contraintes par un invariant et une initialisation : $\mathcal{W} = \langle T, V, type, Inv, Init \rangle$ où T est un ensemble de types, V un ensemble de variables, $type : V \rightarrow T$ une fonction qui définit le type de chaque variable, Inv un invariant défini sur V et $Init$ l'initialisation des variables de V.*

L'espace d'états défini ci-dessus est utilisé pour spécifier aussi bien les composants que les services. Dans la suite, $\mathcal{N}$ est un ensemble fini de noms (*names*) et $\mathcal{M}$ est un ensemble fini des noms de *message* avec $\mathcal{M} \subseteq \mathcal{N}$.

3.3 Composants

Un *composant* Kmelia est défini par un espace d'états et un ensemble de services accessibles par leurs noms. On distingue les *services offerts* qui réalisent des

fonctionnalités et les *services requis* qui déclarent les besoins du composant. L'interface du composant liste les services (offerts et requis) déclarés accessibles par le concepteur du composant. Elle doit être cohérente avec la description des services c'est-à-dire : (1) les services offerts et requis ont des noms distincts et (2) les services de l'interface du composant sont un sous-ensemble des services décrits dans le composant. La figure 3.3 montre un extrait du métamodèle d'un composant Kmelia.

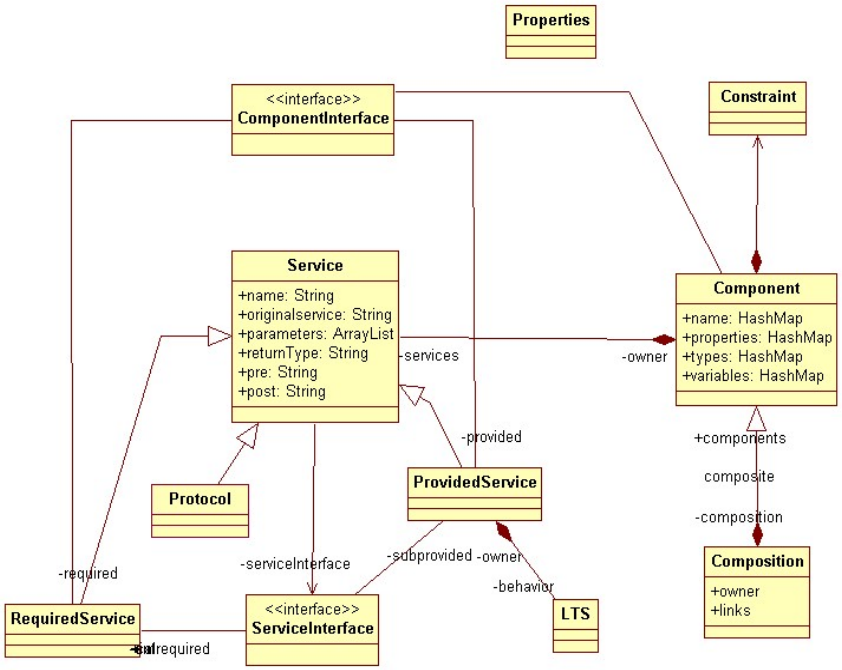


FIGURE 3.3 – Méta-modèle d'un composant Kmelia.

Définition 3.2 (Composant) *Un composant C est un quadruplet $\langle \mathcal{W}, \mathcal{I}, \mathcal{D}, \nu \rangle$ où :*

- *$\mathcal{W}$ l'espace d'états du composant selon la définition 3.1.*
- *$\mathcal{I} \subseteq \mathcal{N}$ l'interface du composant, partitionnée en deux ensembles finis disjoints $\mathcal{I} = \mathcal{I}^P \uplus \mathcal{I}^R$ où $\mathcal{I}^P$ correspond à l'ensemble des noms de services offerts (provided), et $\mathcal{I}^R$ à l'ensemble des noms des services requis (required).*
- *$\mathcal{D}$ est l'ensemble des descriptions de services (cf. section 3.4) qui inclut les services offerts ($\mathcal{D}^P$) et les services requis ($\mathcal{D}^R$) en partition ($\mathcal{D} = \mathcal{D}^P \uplus \mathcal{D}^R$) conforme à celle de $\mathcal{I}$, c'est-à-dire :*
  (1) $\mathrm{dom}(\nu \rhd \mathcal{D}^P) \cap \mathrm{dom}(\nu \rhd \mathcal{D}^R) = \emptyset$
  (2) $\mathcal{I}^P \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P) \wedge \mathcal{I}^R \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^R)$.
- *ν est une fonction qui associe des descriptions aux noms de services ($\nu : \mathcal{N} \rightarrowtail\!\!\!\!\!\rightarrow \mathcal{D}$).*

Pour détailler un peu plus le langage, nous nous appuyons sur le composant *CashDeskApplication*. L'état du composant est décrit dans le listing ci-dessous (listing 3.1).

Listing 3.1 – Spécification Kmelia du composant CashDeskApplication

```
COMPONENT CashDeskApplication
INTERFACE
    provides : {process_sale , register_sale}
    requires : {get_product_info, set_transaction, scan_card}
USES {COCOMELIB}
TYPES
   SALE_STATE :: enum {open, close}
CONSTANTS
    isnull : Integer := 0
VARIABLES
 obs  list_id : setOf Integer;
  obs state : SALE_STATE
  # ...
SERVICES #———— services offerts ————
provided process_sale(id : Integer) : Boolean
  //...
provided register_sale(item : OrderTO; prod : ProductTO) : Boolean
  //...
#———— services requis ————
required get_product_info(prod_id : Integer) : ProductTO
End
// ...
END_SERVICES
```

L'élément-clé de la définition des composants Kmelia est la notion de service que nous détaillons dans la section suivante.

3.4 Services

La notion de service est centrale dans Kmelia. Les services sont au cœur de la construction des composants, de leur composition et des interactions entre eux. Un *service* modélise une fonctionnalité élémentaire ou complexe. Outre sa signature, un service est lui-même constitué d'une interface contractuelle (contenant un ensemble de services dont il dépend et un contrat sous forme de pré/post-conditions), d'un espace d'états, et éventuellement d'un comportement dynamique (un service requis ne définit pas de comportement dynamique). Un *service interne* d'un composant est un service offert qui n'est pas dans l'interface du composant. Il est appelable par un autre service du même composant. Nous qualifions de *sous-service* un service offert d'un composant déclaré dans l'interface d'un autre service du même composant et appelable au sein de ce service.

Définition 3.3 (Service) *Un service s d'un composant C est défini par un quadruplet $\langle \sigma, \mathcal{IS}, \mathcal{W}^L, \mathcal{B} \rangle$ où*

- *$\sigma = \langle s, param, ptype, T, res \rangle$ est la signature du service dans laquelle s est le nom du service, $param$ un ensemble de paramètres, $ptype : param \rightarrow T$ une fonction de typage des paramètres et $res \in T$ le résultat du service ;*
- *$\mathcal{IS}$ est l'interface du service, détaillée ci-après ;*
- *$\mathcal{W}^L$ est l'espace d'états local (cf. définition 3.1) utilisé uniquement dans le comportement dynamique $\mathcal{B}$. Dans le cas d'un service requis, cet espace d'états est appelé* contexte virtuel. *Nous détaillerons ce point dans la section 4.4.2 ;*

– *$\mathcal{B}$ est le comportement dynamique défini sous forme d'un système de transitions étendu (extended labelled transition system -eLTS-) (cf. section 3.4.2).*

3.4.1 Interface du service

Un service Kmelia est une entité de première classe dont le déroulement peut donner lieu à un échange complexe de données entre l'appelant et l'appelé. Par « complexe », nous entendons un échange qui ne se résume pas au passage de paramètres et au retour de l'appel.

> ✎ **Note 8.**
>
> De ce fait, nous ne limitons pas la description d'un service à sa *signature* mais nous l'étendons à la définition d'un contrat sous forme de *pré-condition* d'appel du service, et de *post-condition* du déroulement du service, ainsi qu'à la description de ses dépendances vis-à-vis des autres services .

Il est donc possible de décrire et de vérifier un service selon quatre niveaux de précision : une signature (nom, paramètres, types), une dépendance, un contrat (pré et post-conditions) et un comportement dynamique.

Définition 3.4 (Interface de service) *L'interface du service $\mathcal{IS}$ est défini par un quadruplet $\langle \mathcal{DI}, Cont, \mu, \mathcal{W}^V \rangle$ où*

– *$\mathcal{DI}$ est la* dépendance du service *(service dependency) ; elle est composée des services dont dépend le service courant.*
– *$Cont = \langle Pre, Post \rangle$ est le contrat de service comprenant Pre la pré-condition et $Post$ la post-condition ;*
– *μ est un ensemble de signatures de messages $\mathcal{M} \leftrightarrow (param \times ptype)$; param et ptype sont définis comme dans la signature du service ;*
– *$\mathcal{W}^V$ est un espace d'états virtuel.*

> ✎ **Note 9.**
>
> Notons que $\mathcal{W}^V$ est un concept que nous avons introduit dans le contexte d'un service requis pour pouvoir poser des hypothèses précises sur son fournisseur (cf. section 4.4.2).

La *dépendance* d'un service s est un quadruplet d'ensembles finis et disjoints de noms de services utilisés dans le cadre du service s : `subprovides`, `calrequires`, `extrequires`, `intrequires`. Un service listé dans `calrequires` doit être fourni par le composant lié au service s. Un service listé dans `extrequires` doit être fourni par un composant lié au service listé (et donc dans l'interface du composant dans lequel s est défini). Un service listé dans `intrequires` doit être fourni par le composant qui définit s [AAA06a].

Définition 3.5 (Dépendance de service) *$\mathcal{DI}$ est un quadruplet $\langle sub, cal, req, int\rangle$ d'ensembles disjoints tels que $sub \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P)$ (resp. $cal \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^R$, $req \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P) \wedge req \subseteq \mathcal{I}^R$, $int \subseteq \mathrm{dom}(\nu \rhd \mathcal{D}^P)$) contient les noms des services offerts (resp. ceux requis de l'appelant, ceux requis de n'importe quel composant, les services internes) dans le cadre d'un appel à s.*

Illustrons ceci par l'exemple du service process_sale du listing 3.2 extrait de la spécification Kmelia du composant CashDeskApplication.

Listing 3.2 – Spécification Kmelia du service process_sale

```
provided process_sale(id : Integer) : Boolean
Interface
   // subprovides : {}
   calrequires : {ask_amount}
   extrequires : {get_bar_code,get_product_info, set_transaction, scan_card}
   intrequires : {register_sale}
Pre
    (id in list_id)&&(state = open)
Variables
   prod_id, total, amount_var, rest : Integer;
   authorisation : Boolean;
   credit_info : String; prod_info : ProductTO;
   order : OrderTO;
   payment_mode : PaymentMode
Behavior
   //------- comportement (eLts) du service
Post
   state = open
```

Le service process_sale par exemple dépend du sous-service ask_amount qui devrait être fourni par le composant appelant le service process_sale (calrequires). Il dépend également des services get_bar_code, get_product_info, scan card qui sont fournis par d'autres composants. Remarquons que ces mêmes services apparaissent nécessairement dans l'interface du composant CashDeskApplication (***extrequires***). Enfin, un service register_sale est également requis pour le déroulement du service process_sale mais, contrairement aux précédents, ce service est offert par le même composant contenant process_sale (***intrequires***). Ces dépendances sont explicitées par des flèches dans la figure 3.2.

3.4.2 Comportement du service

Le comportement $\mathcal{B}$ d'un service s est un système de transitions étendu (*extended labelled transition system* -eLTS-). L'extension porte sur l'utilisation d'annotations d'états et de transitions, une annotation étant une référence à un sous-service offert (cf. section 3.4.3).

Définition 3.6 (Comportement du service) *Le comportement $\mathcal{B}$ d'un service, est un eLTS défini par un 6-uplet $\mathcal{B} = \langle S, L, \delta, \Phi, S_0, S_F\rangle$ où*

- *S est l'ensemble des états du service s ;*
- *L est un ensemble d'étiquettes de transitions ;*
- *δ est la relation de transition ($\delta \in S \times L \to S$) ;*

- S_0 *est l'état initial* ($S_0 \in S$) ;
- S_F *l'ensemble fini des états finaux* ($S_F \subseteq S$) ;
- Φ *est la fonction d'annotation d'états* ($\Phi \in S \rightarrow sub_s$).

La figure 3.4 montre le métamodèle d'un service Kmelia. Elle met en évidence la partie comportement détaillée dans cette section. Notons que ce métamodèle sera enrichi pour prendre en compte les aspects fonctionnels tels que les pré/post-conditions (cf. chapitre suivant).

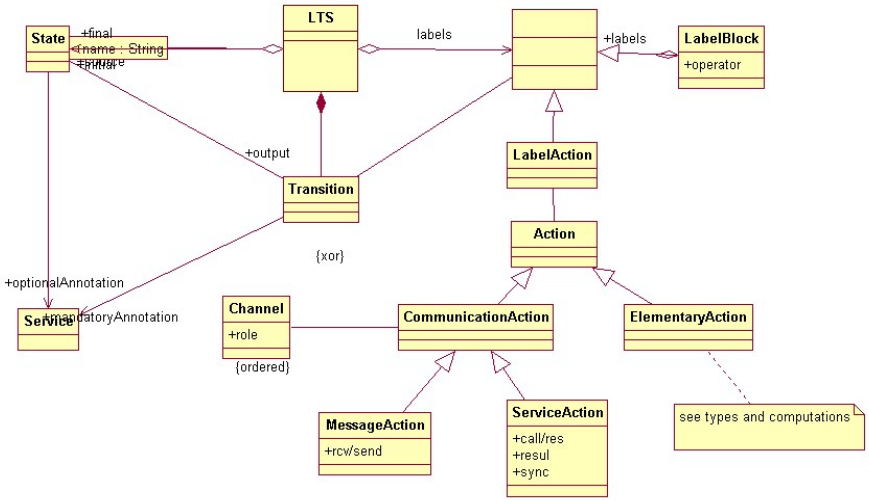


FIGURE 3.4 – Métamodèle d'un service Kmelia.

Dans la figure 3.4, une *étiquette* (LabelAction) est une combinaison, éventuellement gardée, d'actions (`[guard] action*`) ou bien une annotation de transitions (cf. section 3.4.3). Une *action* est une expression Kmelia. Les actions sont soit des *actions élémentaires* soit des *actions de communication* (pour appeler ou terminer un service, pour envoyer ou recevoir un message). Une *action de communication* implique un échange entre deux services, c'est-à-dire un *appel/retour de service* ou un *envoi/réception de message*. Une *action élémentaire*, par exemple une affectation, est une expression qui n'implique pas d'autres services et n'utilise donc pas de canaux de communication. La syntaxe des actions de communication est inspirée du langage CSP de Hoare [Hoa83] : *channel(!| ?| !!| ? ?) message(param*)*. Un *canal de communication* désigne un lien vers un service de la dépendance $\mathcal{DI}$ du service s. Parmi les noms de canaux, *__CALLER* désigne le service appelant, *__SELF* désigne un service du même composant, *_req* désigne le service requis *req*. Par exemple, le comportement du service *process_sale* est représenté dans la figure 3.5.

3.4.3 Composition de services

La composition verticale de services repose sur l'ajout de *points d'expansion* (aux états ou aux transitions du comportement $\mathcal{B}$) qui permettent d'inclure un service dans un autre et qui sont expansés lors des vérifications de compatibilité de services. Un appel de sous-service au sein d'un service s est évalué dans le contexte de s. Ce

FIGURE 3.5 – Comportement dynamique du service process_sale.

sous-service doit être listé dans la dépendance `subprovides` de s. Deux types de points d'expansion sont utilisés en fonction du caractère *obligatoire* ou *optionnel* de l'appel de service désiré. Tout se passe dans ces deux cas comme si on descendait en profondeur dans les systèmes de transition par rapport à un nœud du système de transition du service initial.

Dans la description textuelle des services, une transition $((e_1,\ label),\ e_2) \in \delta$ du eLTS du service est notée e1 −− label −−> e2. Un exemple est donné avec le cas CoCoME pour la description du service de vente *sell* du listing 3.3.

amount est un sous-service de sell, c'est-à-dire un service offert dans le cadre de sell. Le sous-service amount peut être invoqué par l'appelant du service sell dans l'état *e14* uniquement (il est donc *optionnel*). A la fin du déroulement de amount, le contrôle revient dans l'état *e14* ; il est possible d'itérer l'appel de *amount*[1].

1. Une invocation *obligatoire* de ce sous-service aurait pris la forme suivante : *e14 − [[amount]] −> e14bis* et les deux dernières lignes de la spécification auraient commencé par *e14bis*

Listing 3.3 – Spécification Kmelia du service Cashier::sell

```
provided sell()
Interface
   subprovides : {amount}
   extrequires : {ask_sale}
Variables # local to the service
   id : Integer;
   mode : PaymentMode;
   money : Integer;
   sale_success : Boolean
Behavior
   Init i
   Final f
{   i --  {display("New sell, please enter your cashier identifier ");
       id := readInt()    # call an internal action
       } --> e10,
   e10 -- _ask_sale!!ask_sale(id) --> e11,
   e11 -- _ask_sale!start_sale() --> e12,
   e12 -- _ask_sale!end_Sale()  --> e13,
   e12 -- _ask_sale!new_code() --> e12,
  i --  {display("please choose your payment mode");
       mode := readPaymentMode()    # call an internal action
       } --> e10,
      e13 -- _ask_sale!payment(mode) --> e14,
   e14 <<amount>>,      # subservice provided in state e4 only
   e14 -- [mode = cash]_ask_sale?rest_amount(money) --> e14,
   e14 -- _ask_sale??ask_sale(sale_success) --> f
}
End
```

La composition de service sert aussi à définir des *protocoles* dans Kmelia, c'est-à-dire des enchaînements licites de services, faisant office de modes d'emploi pour les composants [AAA07b].

3.5 Composition de composants

Partant d'un ensemble de composants, on peut encapsuler des composants dans d'autres (*composition verticale*) ; ou en assembler avec d'autres (*composition horizontale*). Avant de détailler ces deux formes de composition, nous introduisons dans la section suivante les notions de lien et de sous-lien dans Kmelia.

3.5.1 Lien et sous-lien

Un lien (ou liaison) est un opérateur simple de connexion qui établit (de manière simplifiée) les canaux de communication désignés dans les services. Comparé à d'autres modèles, nous n'introduisons pas de ports ni de connecteurs (cf. section 2.5.1).

Soit $\mathcal{C}$ un ensemble d'*instances* de composant $c_k : C_k$ pour $k \in 1..n$ avec $C_k = \langle \mathcal{W}_k, \mathcal{I}_k, \mathcal{D}_k, \nu_k \rangle$ comme défini dans la section 3.3.

Définition 3.7 (Lien) *Un* lien *entre deux services est défini par un quadruplet de noms de composants et de services avec les restrictions suivantes : (1) les noms de services sont ceux de services définis dans leur composant, (2) un service n'est pas*

lié à lui-même.

$$BaseLink \subseteq (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N})$$
$$(1) \quad \forall (c_i, n_1, c_j, n_2) : BaseLink \bullet n_1 \in \operatorname{dom} \nu_i \wedge n_2 \in \operatorname{dom} \nu_j$$
$$(2) \quad \forall c_i : \mathcal{C},\ n_1 : \operatorname{dom} \nu_i \bullet (c_i, n_1, c_i, n_1) \notin BaseLink$$

où $\operatorname{dom} \nu_i$ *est le domaine de la fonction de nommage des services, c'est-à-dire l'ensemble des noms de services du composant* C_i*.*

Un *sous-lien* est un lien défini dans le contexte d'un autre lien. La notion de sous-lien est relative à la dépendance de service sur l'appelant (l'ensemble *cal* de $\mathcal{DI}$) et les sous-services (l'ensemble *sub* de $\mathcal{DI}$). Il n'y a pas de dépendance circulaire entre liens (3).

$$SubLink : BaseLink \leftrightarrow BaseLink$$
$$(3) \quad \forall (l_1, l_2) \in SubLink \bullet (l_2, l_1) \notin SubLink^*$$

Où $SubLink^*$ est la fermeture transitive (non réflexive) de la relation $SubLink$.

Dans Kmelia, les liens et les sous-liens apparaissent sous forme de *liens d'assemblage* dans un *assemblage* et de *liens de promotion* dans un *composite*. Nous détaillons respectivement ces deux formes dans les deux sections suivantes.

3.5.2 Assemblages

Un *assemblage* Kmelia est un ensemble de composants liés sur leurs services par des *liens d'assemblage*. Pour plus de *flexibilité*, le modèle Kmelia permet l'assemblage partiel de composants et le renommage des composants et des services dans un assemblage. Il n'est pas nécessaire de lier tous les services offerts et tous les services requis dans un assemblage, à condition toutefois que ces services ne soient pas invoqués. On peut ainsi disposer d'un composant riche et complexe mais ne l'utiliser que partiellement pour ne pas avoir à développer (et donc à vérifier) un composant spécifique. Le renommage des composants et des services permet de les adapter au contexte d'une application en les rendant plus lisibles.

Définition 3.8 (Assemblage) *Un assemblage de composants est un quintuplet* $A = \langle \mathcal{C}, alinks, subs, vmap, mmap \rangle$ *où :*

- $\mathcal{C}$ *est un ensemble d'instances de composant* $c_k : C_k$ *pour* $k \in 1..n$*,*
- *alinks est un ensemble de liens d'assemblage entre services de* $\mathcal{C}$*,*
- *subs est une relation d'inclusion de liens,*
- *vmap est une fonction de correspondance de variables d'état,*
- *mmap est une fonction de correspondance de messages.*

> ✎ **Note 10.**
>
> Les fonctions de correspondance *vmap* et *mmap* sont deux concepts que nous avons introduits dans le cadre d'un assemblage de composants pour établir explicitement la relation entre le contexte virtuel du service requis et le contexte réel du composant du service offert. Elles seront détaillées dans le chapitre suivant (cf. section 4.4.2).

Dans la suite, nous détaillons les liens *alinks* et les sous-liens *subs*.

3.5.2.1 Liens et sous-liens d'assemblage

Un *lien d'assemblage* entre un service n_1 d'un composant C_1 et un service n_2 d'un composant C_2 est une abstraction d'un canal de communication reliant n_1 à n_2. Chacun des services n_1 et n_2 est listé dans l'interface du composant le contenant.

$$
\begin{array}{ll}
& alinks \subseteq BaseLink \wedge \\
(1) & (\forall (c_i, n_1, c_j, n_2) : alinks \bullet c_i \in \mathcal{C} \wedge c_j \in \mathcal{C} \wedge \\
(2) & \quad ((n_1 \in \mathcal{I}_i^P \wedge n_2 \in \mathcal{I}_j^R) \vee (n_1 \in \mathcal{I}_i^R \wedge n_2 \in \mathcal{I}_j^P))) \\
& subs \subseteq SubLink \wedge \\
(3) & (\mathrm{dom}\, subs - \mathrm{ran}\, subs) \subseteq alinks \wedge \\
(4) & (\forall ((c_i, n_1, c_j, n_2) \mapsto (c_k, n_3, c_l, n_4)) \in subs \bullet c_i = c_k \wedge c_j = c_l) \wedge \\
(5) & (\forall (c_i, n_1, c_j, n_2) : \mathrm{ran}\, subs \bullet ((\nu_i(n_1) \in {\mathcal{D}^P}_i) \text{ xor } (\nu_j(n_2) \in {\mathcal{D}^P}_j)))
\end{array}
$$

Les contraintes ont la signification suivante : les composants des liens sont les composants de l'assemblage (1). Il y a symétrie *requis-offert* dans un lien (2). Les sous-liens dépendent *in-fine* des liens d'assemblage de plus haut niveau (3) et concernent les mêmes composants (4). Les services offerts sont liés aux services requis (2 et 5). Les liens établissent donc des ponts de nommage explicites entre services ; pour avoir des assemblages complètement définis, on doit aussi faire correspondre les espaces d'état et les messages.

Un lien d'assemblage est structuré en concordance avec les dépendances de services (cf. section 3.4.1). Si les services établissent des dépendances de type *subprovides* et *calrequires*, alors des sous-liens sont à définir dans le cadre de ces services. Leur spécification est similaire à celle des liens.

3.5.2.2 Illustration

Reprenons l'exemple CoCoME. La figure 3.6 et le listing 3.4 représentent un assemblage partiel du système de gestion des ventes *Trading System* qui enrichit celui de la figure 3.2 avec les composants de gestion de persistance *Inventory*, du consortium bancaire *Bank* et de l'interface *Cashier*.

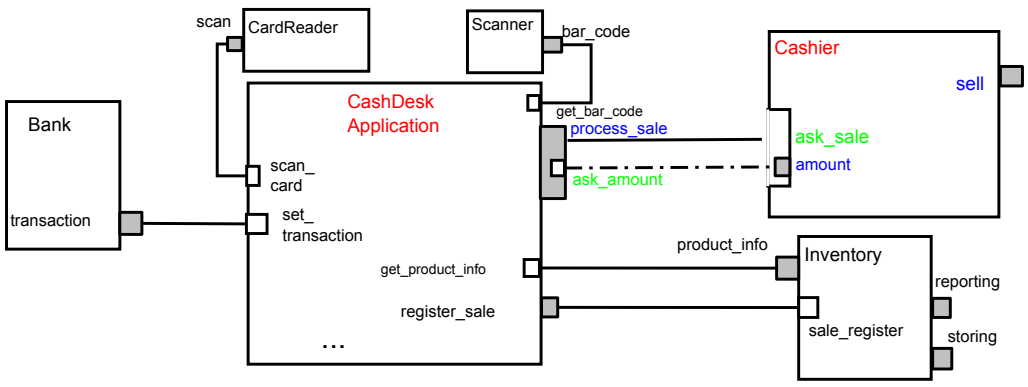


FIGURE 3.6 – Description Kmelia simplifiée d'un assemblage pour le cas CoCoME.

Dans la figure 3.6, les traits pleins entre services représentent les *liens d'assemblage* qui associent des services offerts à des services requis (un service requis est « réalisé » par un service offert). Les traits discontinus sont des sous-liens. Les assertions des services (pré/post-conditions) définissent des contrats qui respectent l'invariant défini dans l'espace d'états de leurs composants (ou l'espace d'états virtuel pour un service requis). Ces assertions sont reprises pour former une partie du contrat d'assemblage (en plus des compatibilités de signature, de sous-liens et d'interactions).

Listing 3.4 – Spécification Kmelia d'un assemblage du système CoCoMe

```
Assembly
  Components
    cr  : CardReader;
    s   : Scanner ;
    cda : CashDeskApplication
    b : Bank;
    c : Cashier
  Links /////////// liens d'assemblage /////////
    @code: r-p cda.get_bar_code s.bar_code
    @scan: r-p cda.scan_card cr.scan
    @trans: r-p cda.set_transaction b.transaction
              message mapping
               set_amount= transaction_amount
    @sale: p-r cda.process_sale c.ask_sale
       sublinks : {lamount}
    @lamount: r-p cda.ask_amount c.amount
    ...
End // assembly
```

Par exemple, les deux composants cda : CashDeskApplication et c : Cashier sont assemblés par un lien d'assemblage nommé @sale indiquant que le service requis ask_sale de *c* est *réalisé* par le service offert process_sale de *cda*. Le préfixe p−r indique le sens de lecture du lien. De même, le sous-lien nommé @lamount exprime une dépendance entre les services process_sale et ask_sale. Ainsi, d'une part, le service process_sale requiert le service ask_amount dans sa dépendance `calrequires`, et d'autre part, le service amount est défini dans la dépendance `subprovides` du service ask_sale. Le sous-lien @lamount établit le lien entre deux sous-services ask−amount et amount.

3.5.2.3 Interaction inter-composants

L'assemblage induit une interaction inter-composants. Considérons dans un premier temps des interactions basées sur des liaisons d'un service/composant à un autre service/composant. L'interaction entre deux services venant de deux composants distincts est la base de l'interaction inter-composants. Un service d'un composant, pendant son déroulement, appelle un autre service qui est un requis du composant. Le déroulement se passe comme si le service requis était remplacé par le service effectif offert par un autre composant. Les deux services se déroulent en parallèle et échangent en communiquant sur un canal abstrait (lien d'assemblage). Les services interagissent via les communications synchrones ou asynchrones.

En se basant sur les actions de communication définies dans la section 3.4.2, une interaction est une communication synchrone établie par une paire complémentaire *envoi de message(!)-réception de message(?), appel de service(!!)-attente du démarrage de service(??), envoi du résultat de service(!!)-attente du résultat de service(??)*. La forme élémentaire des actions de communication a été étendue pour exprimer diverses autres interactions comme nous le verrons dans la section 3.5.2.5.

Un service pouvant communiquer avec plusieurs autres services de composants différents, l'interaction entre composants se généralise à une interaction entre plusieurs composants via le comportement global formé par les services communicants. En effet, le déroulement d'un service peut entraîner en cascade le déroulement d'autres services et donc la communication avec eux.

3.5.2.4 Assemblage multiple

Dans ce qui précède, nous avons considéré des assemblages simples : (1) chaque instance d'un composant est nommée par une variable, (2) les liens sont exclusivement entre deux composants (de type `1-1`). L'exclusivité du lien signifie que si `n` services sont liés à un service, alors on considère réellement `n` liens indépendants. Cependant, le modèle a été étendu dans [AAA08] pour autoriser des interactions multi-parties. Cette extension répond à un besoin de modélisation des applications impliquant plusieurs parties, par exemple un système d'enchères ou un système de *chat* avec un modérateur et des participants.

Dans une composition multi-parties, on autorise plusieurs instances d'un composant (sous forme de tableau de composants) et des *services partagés* pour permettre les liens `1-n` ou `n-1`. Un service partagé est un service utilisé simultanément, de manière transparente, par plusieurs autres services. Cette extension met en œuvre des primitives de communication multi-parties (cf. section 3.5.2.5). Un tableau de composants de même type C est déclaré par c[n]:C, où n est le nombre maximum d'exemplaires de composants. Chaque composant est accessible par son indice dans le tableau c[i].

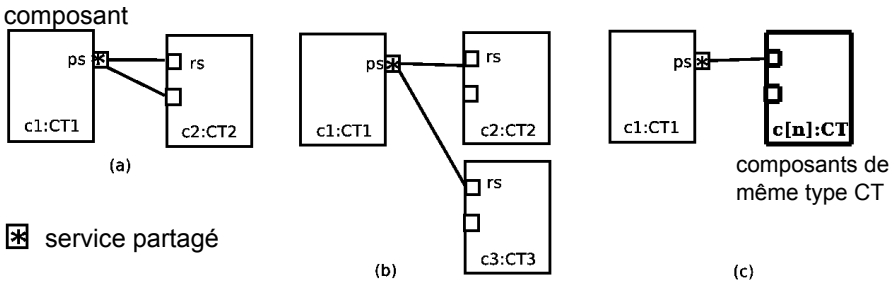


FIGURE 3.7 – Différents types d'assemblage avec des services offerts partagés

Un *service offert partagé* est un service lié à plusieurs services requis de plusieurs composants de types quelconques (lien de type `1-n`). Par exemple, un service de *chat* en conférence est invoqué par plusieurs participants. Par défaut, les appelants sont rangés dans un tableau, ils ont le même rôle dans la collaboration mais il est possible de leur faire jouer des rôles différents. Dans le *chat*, l'initiateur a des prérogatives.

Les appelants n'ont *a priori* aucune connaissance les uns des autres.

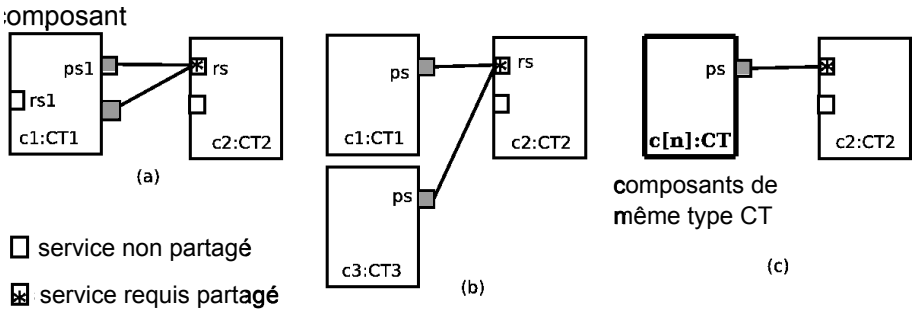


FIGURE 3.8 – Différents types d'assemblage avec des services requis partagés

Un *service requis partagé* est un service lié à plusieurs services offerts de plusieurs composants de types quelconques. Il va donc mettre en concurrence plusieurs fournisseurs. Par exemple, un service *Web* de comparatif peut émettre des requêtes et collecter les résultats de différentes façons (le premier reçu, le meilleur selon un critère donné, *etc.*).

3.5.2.5 Interaction multi-parties

Lorsqu'il existe des liens entre un service d'un composant et plusieurs services d'un ou plusieurs composants, il y a potentiellement une interaction entre plusieurs services. On est dans le cadre d'une généralisation des interactions entre services. Les communications et les primitives associées sont adaptées en conséquence. Dans ce contexte, les actions de communication prennent la forme suivante :

`channel[<selector>](!|?|!!|??)message(param*)`

Les valeurs de `<selector>` sont : `ALL` pour tous les services concernés par la communication, `i` pour un service donné et `:i` pour un service quelconque.

Lorsqu'un service offert `servP` est partagé par plusieurs services requis `servR_i`, il y a interaction simultanée entre le comportement du service `servP` et les comportements respectifs des services `ServR_i`. Cette interaction se déroule comme suit : (1) les actions élémentaires de tous les services se déroulent de façon indépendante, et sont par conséquent entrelacées. (2) les actions de communication utilisées dans les comportements des services forcent soit à une synchronisation entre tous les services (c'est le cas quand les opérateurs sont précédés de sélecteurs spécifiques [ALL]?? ou [ALL]!!, soit à une synchronisation entre certains services uniquement (c'est le cas avec [:i]?? ou [:i]!!). De la même manière que dans le cas précédent, lorsqu'un service requis partagé est lié à plusieurs services offerts, il y a interaction multiple simultanément ; les actions élémentaires sont entrelacées alors que celles de communication sont régies par la sémantique des opérateurs de communication.

3.5.3 Composites

La *composition verticale* de composants consiste en l'encapsulation d'un assemblage dans un composant appelé *composite*. La figure 3.9 présente un extrait du métamodèle d'une composition Kmelia. Ainsi une succession de compositions verti-

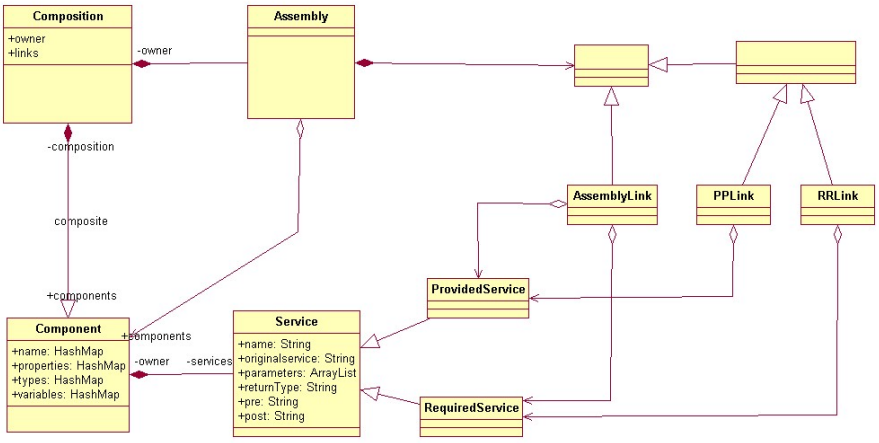


FIGURE 3.9 – Méta-modèle d'une composition de composants Kmelia.

cales aboutit à une hiérarchie de composants. L'encapsulation pouvant se faire de manière répétée, la visibilité des espaces d'état se fait de proche en proche. De la même façon, les variables et les services ainsi que leurs dépendances, peuvent être promus à l'interface du composite. Nous reviendrons sur ces notions dans le chapitre suivant une fois introduit l'enrichissement du modèle Kmelia en terme de langage de données.

Définition 3.9 (Composite) *Un composite est un quadruplet* $CC = \langle C, \mathcal{A}, plinks, pvars \rangle$ *où :*

- $C = \langle \mathcal{W}, \mathcal{I}, \mathcal{D}, \nu \rangle$ *est un composant (cf. section 3.3). Le composite redéfini par défaut est SELF : C.*
- $\mathcal{A} = \langle \mathcal{C}, alinks, subs, vmap, mmap \rangle$ *est un assemblage (cf. section 3.5.2).*
- *plinks est un ensemble de liens de promotion entre les services de* $\mathcal{A}$ *qui ne sont pas déjà liés dans alinks :*

$$
\begin{array}{ll}
& plinks \subseteq BaseLink \wedge (\forall (c_i, n_1, c_j, n_2) : plinks \bullet \\
(1) & \quad c_i \in \mathcal{C} \wedge c_j = SELF \wedge \\
(2) & \quad ((n_1 \in \mathcal{I}_j^R \wedge n_2 \in \operatorname{dom} \nu^R) \vee (n_1 \in \mathcal{I}_j^P \wedge n_2 \in \operatorname{dom} \nu^P)) \wedge \\
(3) & \quad (\forall c_k \in \mathcal{C}, n_3 \in \mathcal{I}_k \bullet (c_i, n_1, c_k, n_3) \notin alinks \wedge (c_k, n_3, c_i, n_1) \notin alinks) \wedge \\
(4) & \quad \nu(n_2) = rename(\nu(n_1), n_2))
\end{array}
$$

  où $\operatorname{dom} \nu$ *est un ensemble de noms des services du composite,* $rename(s, n)$ *est une fonction qui renvoie un service renommé* n *au lieu de* s*.*
- *pvars est une fonction associée à chaque lien de promotion (cf. section 4.4.2).*

Notons que la spécification d'un composite ne contient pas les spécifications de ses sous-composants mais elle y fait référence. Un lien de promotion associe le composant SELF à un des composants de l'assemblage (1). Le service promu garde sa nature (*provided* ou *required*) (2). Les services promus ne sont pas déjà liés dans un assemblage (3). La définition d'un service promu est la même que celle du sous-composant à un renommage près (4).

Afin de faciliter la réutilisation et le passage à l'échelle, l'encapsulation d'un assemblage dans un composite masque par défaut le détail des composants internes. Cependant, grâce à la notion de promotion, le composite peut utiliser directement certaines variables de ses composants internes ainsi que certains services définis dans leurs interfaces.

3.5.3.1 Illustration

En guise d'illustration, prenons l'architecture du système CoCoME de la figure 3.10 qui restructure l'assemblage de la figure 3.6. Les emboîtements de composants dénotent la relation de composition par encapsulation de composants ; les traits doubles sur les services sont des *liens de promotion* ; un service d'un composant est promu au niveau du composite qui le contient.

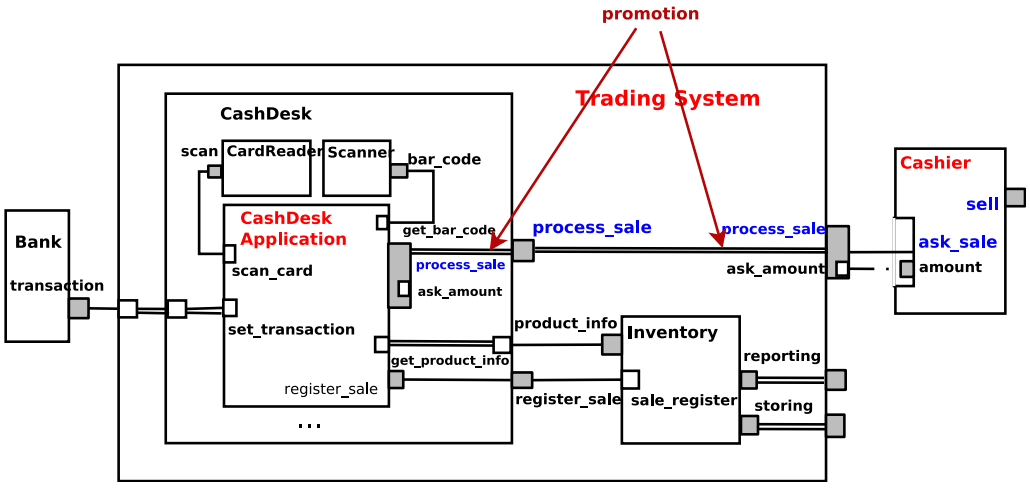


FIGURE 3.10 – Description Kmelia simplifiée d'un assemblage pour le cas CoCoME.

Dans la spécification textuelle, les liens de promotion sont une section de la *Promotion*. Par exemple, dans le listing 3.5, le service offert `process_sale` du composant `cda : CashDeskApplication` est promu au niveau du composite `CashDesk`, dont il devient un point d'entrée.

La promotion de la variable `state : SALE_STATE` du composant `cda` se fait dans la déclaration de l'espace d'états du composite `CashDesk` par `status : SALE_STATE FROM cad.state`. Le renommage est facultatif. Le principe d'encapsulation est appliqué strictement : le composite n'a pas plus de droits qu'un composant client lié par assemblage ; pour modifier l'état d'un composant, il doit passer par les services de ce composant.

D'un point de vue méthodologique, l'indépendance composant/composite permet aussi bien une approche descendante qu'ascendante des systèmes à composants. L'évolution se fait alors en utilisant des règles précises de conformité de la promotion. Nous étudierons ce point plus en détail dans le chapitre 5 (section 5.2.2.1).

Listing 3.5 – Spécification Kmelia du composite CashDesk

```
COMPONENT CashDesk
INTERFACE
  provides : {process_sale, register_sale}
  requires : {set_transaction, get_product_info}
COMPOSITION
  Assembly
    Components
      cr : CardReader;     s : Scanner ;
      cda : CashDeskApplication
    Links //////////// assembly links /////////
      @code: r-p cda.get_bar_code s.bar_code
      @scan: r-p cda.scan_card cr.scan
  End // assembly
  Promotion
    Links //////////// promotion links /////////
      @sale: p-p cda.process_sale SELF.process_sale
         sublinks : {lamount}
      @regis: p-p cda.register_sale SELF.register_sale
      @trans: r-r cda.set_transaction SELF.set_transaction
      @prod: r-r cda.get_product_info SELF.get_product_info
         //////////// promotion sublinks /////////
      @lamount: r-r cda.ask_amount SELF.ask_amount
END_COMPOSITION
```

3.5.3.2 Interaction intra-composant

Nous précisons ici, d'une part, la manière dont les services, les services internes et les sous-services d'un même composant interagissent et, d'autre part, la manière dont les services de composants imbriqués interagissent.

- Considérons l'interaction entre un service et un de ses sous-services. Un service d'un composant C et ses sous-services ne partagent pas leurs variables locales, mais celles du composant C. Un service peut, lors de son déroulement, invoquer un sous-service. Le sous-service prolonge alors le déroulement de son appelant, il n'y a pas d'interaction entre eux. Un service peut communiquer avec ses sous-services à travers les variables du composant. L'appel/retour de service crée/termine cette interaction entre un service et un sous-service dans un composant.
- En ce qui concerne les services internes, il peut y avoir une interaction entre un service interne et son appelant ; les interactions sont ici basées sur les canaux de communication abstraits associés aux services. Les opérateurs d'appel/retour de service (`!!` et `??`) créent/terminent cette interaction entre un service et un service interne dans un composant.
- Lorsqu'un composant `C` est encapsulé dans un composite `CC`, pour utiliser un service de `C`, il faut le promouvoir et ensuite l'utiliser comme un service interne.

3.6 Vérification de propriétés dans Kmelia

Dans cette section, nous nous intéressons à la vérification des propriétés dans le modèle Kmelia. Nous décomposons la vérification des spécifications Kmelia en deux

parties : la vérification des composants individuels et la vérification de leur composition (assemblages et composites). La première est liée à la cohérence interne des composants. La seconde atteste que, mis ensemble, les composants restent corrects. La vérification des propriétés se décompose en une collection de propriétés structurelles, fonctionnelles et comportementales. Dans la suite, nous faisons un survol des propriétés attendues, quelques analyses effectuées sur des composants Kmelia et leur composition ainsi que la mise en œuvre de ces dernières.

3.6.1 Propriétés globales à vérifier

Nous abordons dans cette section les possibilités d'étude offertes par la disponibilité d'un modèle formel [AAA+04]. La correction d'un composant, élémentaire ou non, est assurée par une vérification formelle des propriétés préalablement exprimées. Par conséquent, l'analyse formelle des composants inclut la vérification des propriétés suivantes :

- les ***propriétés d'interopérabilité statique*** : elles concernent la compatibilité des signatures et des interfaces des composants et de leurs services (nommage et typage) ; nous considérons aussi le fait qu'un composant donne suffisamment d'informations sur son interface afin d'être (ré)utilisé par d'autres composants ;
- les ***propriétés architecturales ou structurelles*** : il s'agit de la disponibilité des composants et des services requis, de la correction des liens entre interfaces de composants (fournisseurs et clients), etc ;
- les ***propriétés d'interopérabilité dynamique*** (compatibilité comportementale) ; il s'agit de la correction des interactions entre les services de deux (ou plusieurs) composants qui sont assemblés. Plusieurs aspects sont considérés : différentes formes d'interactions, communication synchrone ou non, atomicité ou non des actions, etc ;
- les ***propriétés de corrections fonctionnelles*** : les composants (et leurs services) font-ils ce qu'ils doivent faire ? Ces propriétés peuvent être vérifiées indépendamment sur les composants élémentaires et également sur la composition des composants ;
- la ***compositionnalité*** est un aspect important : les propriétés d'un système global doivent pouvoir être inférées à partir des propriétés locales des composants qui constituent le système ;
- la ***facilité de la maintenance*** (facilité de modification, évolution des composants et des applications qui les utilisent). Les composants doivent pouvoir être mis à jour simplement sans causer de dommage aux composants tiers qui les utilisent. La mise à jour d'un composant inclut : la modification de l'implantation de ses services, l'ajout et/ou la suppression de services, etc ;
- l'***hétérogénéité*** : dans le contexte du CBSE, les composants provenant de divers fournisseurs doivent pouvoir être composés pour développer de très grandes applications. Cet aspect est un défi important puisque les composants peuvent par exemple avoir des modèles différents.

Dans Kmelia, les concepts de service, de composant, d'assemblage et de composite sont analysés selon différentes facettes, telles que la cohérence, la correction,

l'intégrité de la communication, l'absence d'interblocage, la vivacité, l'atteignabilité, la compatibilité des liens d'assemblages ou de promotion, le respect des contrats, *etc.*

Les composants sont analysés par rapport à des propriétés ; on peut ainsi vérifier des propriétés inhérentes aux modèles, telle que la cohérence des interfaces de composants et de services ; la terminaison des services, la préservation des invariants par les services, la correction fonctionnelle du service (les assertions pré/post *vs.* le comportement du service), *etc.*

Les assemblages sont essentiellement analysés au niveau de la composabilité des liens qui, outre la concordance des sous-liens avec les interfaces de services, se matérialise sur une échelle à quatre niveaux par la cohérence des signatures, des interfaces de services, des contrats de services et par la compatibilité comportementale. La cohérence des contrats implique de prouver les implications d'assertions. La compatibilité comportementale est évaluée lien par lien pour limiter l'explosion combinatoire, elle comprend l'évaluation des synchronisations pour déterminer la terminaison des services. On peut aussi analyser la continuité des services (toutes les dépendances d'un service offert sont-elles transitivement résolues ou pas), *etc.*

Les composites sont analysés d'abord en tant que composants. Ensuite les propriétés relatives à la promotion (règles de visibilité, respect des contrats à travers la promotion, *etc.*) peuvent être vérifiées.

Le tableau 3.6.1 présente un aperçu des propriétés à vérifier pour valider les spécifications Kmelia ainsi que leur statut en terme d'outillage automatique.

	Niveau de vérification							
Propriétés	Service		Composant		Assemblage		Composite	
Contrôle de type		✓		✓		✓		✓
Portée de variables		✓		✓		✓		✗
Règles d'observabilité	prédicats *Pre/Post*	✗	variables, invariant	✗	pré/post, invariant, variables, *context mapping*	✗	pré/post, invariant, variables	✗
Dépendances de services	$\mathcal{DI}$ *vs.* $\mathcal{B}$ (eLTS)	✓	$\mathcal{DI}$ *vs.* $\mathcal{I}$ et $\mathcal{D}$	✓	disponibilité des services	✓	$\mathcal{DI}$ *vs.* $\mathcal{I}$ et $\mathcal{D}$	✓
Correction	compatibilité de signatures	✓			compatibilité de signatures	✓		
	contexte virtuel (service requis)	✗	préservation de l'invariant	✗	conformité des contrats	✗	préservation de l'invariant	✗
	Pre/Post vs. $\mathcal{B}$ (eLTS)	✗	protocole *vs.* *Pre/Post*	✗			sûreté de la promotion	✗
Dynamique	absence d'interblocage	✓			compatibilité comportementale	✓	conformité comportementale	✗

✓ : Propriétés vérifiées dans la plate-forme COSTO.
✗ : Propriétés non encore supportées par la plate-forme COSTO et faisant l'objet de ces travaux de thèse.

Tableau 3.6.1 – Quelques propriétés à vérifier sur une spécification Kmelia

3.6.2 La composabilité

Une propriété doit d'abord être formellement exprimée pour être prouvée. Dans cette section, nous définissons la propriété de *composabilité* de composants [AAA06c]. La composabilité est définie essentiellement sur les services et généralisée aux composants. De manière informelle, les services des composants sont composables si on peut les lier pour créer un assemblage de telle sorte que les services interagissent. Le modèle Kmelia nous a servi de base de raisonnement pour l'analyse de la *composa-*

bilité sur des compositions. La composabilité se rapporte à la compatibilité statique et l'interopérabilité dynamique (compatibilité comportementale).

Définition 3.10 (Composabilité) *Un composant C_i est dit composable dans un assemblage pour un ensemble de services offerts S_o si :*

1. *chaque service offert $s \in S_o$ est complet vis-à-vis de ses services requis $sr(s)$, c'est-à-dire que les $sr(s)$ sont liés à des services offerts ;*
2. *chaque service offert $s \in S_o$ est compatible,* au sens de la définition 3.11*, avec les services qui le requièrent.*

La *composabilité* d'un composant dans un assemblage est fondée non seulement sur la compatibilité des signatures contenues dans les interfaces des services impliqués mais aussi sur les comportements des services. Cette définition prend en compte le fait que certains services inutilisés dans un assemblage ne sont pas à vérifier. Il n'est, en effet, pas utile de vérifier la compatibilité avec les services requis $sr(s)$ dans le composant C_i parce que cela est fait lors de l'étude de la composabilité des composants offrant les services de $sr(s)$.

La *composabilité totale* est obtenue par l'union des composabilités partielles : si toutes les vérifications deux-à-deux fonctionnent, alors elles fonctionnent globalement. Par conséquent, l'interaction globale se réduit à une suite d'interactions locales [AL03]. En pratique, nous nous basons sur ceci pour vérifier la composabilité.

Définition 3.11 (Compatibilité) *Un service offert so_{C_i} d'un composant C_i est compatible avec un service requis sr_{C_j} d'un composant C_j, avec lequel so_{C_i} est lié dans un assemblage si :*

1. *les interfaces des services sont compatibles, c'est-à-dire si :*
   (a) *les signatures concordent (règles de typage),*
   (b) *les assertions sont cohérentes (pré/post-conditions) et*
   (c) *les liaisons (liens, sous-liens) et les signatures enrichies sont conformes (concordance des signatures et cohérence des assertions des sous-services).*
2. *pour chaque service offert so_{C_j} du composant C_j qui invoque sr_{C_j}, les interactions entre les comportements de so_{C_i} et so_{C_j} (y compris leurs sous-services) sont compatibles. La compatibilité se traduit ici par l'absence de blocage.*

Le contexte d'une interaction est défini par le triplet $< so_{C_j}, sr_{C_j}, so_{C_i} >$ (c'est-à-dire `<service appelant, service requis par l'appelant, service offert>`). Le service offert est lié au service requis dans l'assemblage. Le déroulement des services et sous-services participant à l'interaction doit aboutir à un état final pour le service offert.

Compatibilité statique Du fait de la description des services, la compatibilité statique lors des assemblages de composants est vérifiée de façon progressive.

Une première étape de la vérification de la compatibilité statique est faite à la compilation de la spécification Kmelia par une analyse des interfaces en profondeur : vérification des signatures de tous les services en jeu, concordance des liens et des

sous-liens, complétude des services requis pour tout service offert dans la composition (seuls sont traités les services effectivement utilisés dans la composition).

A ce stade, certaines incompatibilités peuvent être détectées. La plupart des propriétés de l'interopérabilité (statique) et des propriétés architecturales est couverte par ces algorithmes.

Lorsque cette première étape de compatibilité est terminée, les assertions des services liés sont utilisées pour s'assurer du respect du contrat d'assemblage.

Compatibilité fonctionnelle Soit un service ServR (d'un composant C) avec les assertions $PreR$ et $PostR$ et ServP un service offert avec les assertions $PreP$ et $PostP$. Le service servP permet de satisfaire le requis ServR lorsque :

$$PreR \Rightarrow PreP \wedge PostP \Rightarrow PostR$$

De cette façon, on décrit un contrat de telle sorte que les composants qui peuvent être assemblés avec C sont ceux qui sont à même de respecter le contrat ci-dessus.

Compatibilité comportementale Pour la vérification de l'interopérabilité dynamique (ou comportementale), on explore l'évolution dynamique des échanges. L'interaction entre composants se traduit par l'interaction entre leurs services. Ainsi nous nous basons sur une analyse de l'interaction pair à pair de services (appelant et appelé sur une liaison).

L'interopérabilité dynamique est alors assurée lorsqu'il n'y a pas de blocage dans les interactions entre les services considérés, interprétés comme des processus. Sur la base de cette similarité, nous avons formalisé la correspondance entre les services des composants et les processus, puis nous avons développé deux outils de traduction de spécifications Kmelia dans des langages adaptés et outillés MEC [AAA06b] et LOTOS/CADP [AAA06c]. L'analyse de la spécification est alors faite avec ces outils. C'est la technique de *model checking* qui est la plus utilisée dans ce cas.

L'intérêt de MEC est sa proximité avec les LTS, la traduction est plus simple et l'interprétation des résultats est immédiate. Malheureusement, il ne permet pas le traitement des gardes et des données. A l'inverse, LOTOS permet un traitement des gardes et des données mais la traduction du LTS d'un service en LOTOS génère un ensemble de processus dont le cardinal est fonction de la complexité et de la taille du eLTS de départ. De fait, la traduction n'est pas directe et l'interprétation des résultats est plus délicate.

3.7 La boîte à outils COSTO

Nous donnons ici un aperçu rapide de la plate-forme logicielle COSTO[1] associée à Kmelia, puis nous présentons quelques aspects des analyses effectuées sur les spécifications Kmelia. Ces analyses sont détaillées dans [AAA07a].

1. COmponent Study TOolkit

La figure 3.11 contient une représentation simplifiée de l'architecture de COSTO [1] dont les principaux modules sont les suivants :

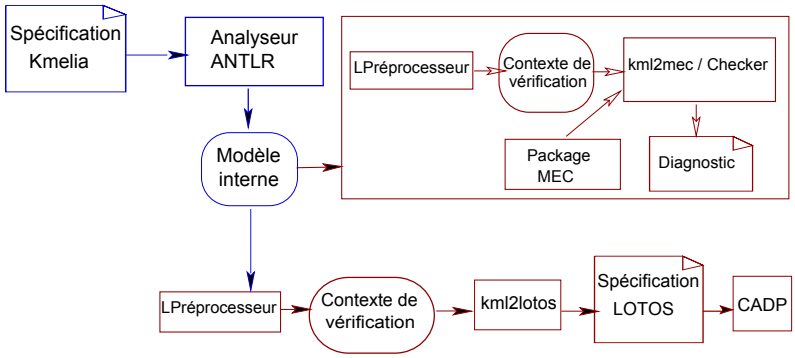


FIGURE 3.11 – Aperçu de l'architecture de COSTO

- un éditeur pour faciliter l'écriture de spécifications Kmelia.
- un analyseur (*Type-checker*) pour les spécifications Kmelia ; l'analyseur est développé avec la technologie Java/Antlr. A l'issue de l'analyse d'une spécification Kmelia, un modèle interne à objets est généré à l'aide d'une librairie entièrement développée pour satisfaire les besoins spécifiques à Kmelia.
- des passerelles vers MEC et LOTOS/CADP ; le but est la vérification des interactions entre les systèmes de transition qui modélisent les services. Ici nous nous sommes appuyés sur des environnements existants afin d'exploiter au mieux les résultats connus en terme d'analyse de la dynamique des systèmes.
- divers utilitaires, pour présenter les spécifications sous forme graphique et pour générer des documents Latex des spécifications.

3.8 Conclusion

Nous avons présenté dans ce chapitre le modèle Kmelia pour la conception et le développement à base de composants. Kmelia est un modèle :

- simple, ne supportant que peu de concepts fondamentaux (service, composant, assemblage et composite) ;
- abstrait, afin de maîtriser la complexité des systèmes spécifiés ;
- extensible, pour pouvoir intégrer d'autres niveaux de contrats ;
- formel, permettant de raisonner sur des propriétés exprimées au préalable ;
- pouvant être raffiné pour tirer profit des plates-formes existantes.

De plus, les aspects formels permettent de mettre en œuvre l'analyse automatique des propriétés dont nous avons donné les principales caractéristiques dans la plate-forme COSTO qui accompagne Kmelia. Nous avons également mis en évidence les parties ouvertes sur lesquelles nous avons contribué.

Dans la suite de cette thèse, nous nous basons sur ce modèle afin de lever certaines limitations des approches à base de composants et services ; nous visons à

1. Il s'agit de son architecture au début de cette thèse. Le chapitre 7 montre une extension que nous avons développée dans le cadre de cette thèse.

simplifier et ainsi à rendre systématique la construction par composants en utilisant les approches formelles dès les premières phases du développement de logiciels. Nous pouvons ainsi construire des composants dont on peut, d'une part, certifier, par des preuves formelles, la correction vis-à-vis des spécifications et qu'on peut, d'autre part, maintenir plus facilement en remontant à leurs spécifications. D'un point de vue méthodologique, notre travail vise aussi à montrer le bien-fondé de l'approche par contrats pour la conception ascendante ou descendante, et l'intérêt de la spécification précise des besoins (au sens des services requis) pour mettre en œuvre le concept de composant sur étagère (COTS pour *component off-the-shelf*).

Chapitre 4

Un langage de données pour exprimer les contrats

Dans ce chapitre, nous présentons un langage de données pour décrire l'aspect fonctionnel de Kmelia qui couvre la définition des types de données, l'état des composants, les expressions, les assertions des services (pré/post-conditions), les actions, les communications, le typage des composants et des assemblages. Les techniques de vérification des contrats seront détaillées dans les deux chapitres suivants.

4.1 Prise en compte des données

4.1.1 Motivations

La prise en compte des données dans les modèles à composants n'est pas nouvelle. Cependant, comme nous l'avons déjà signalé dans le chapitre 2, les modèles opérationnels tels que Corba, EJB ou .NET ne permettent pas de raisonner au niveau qui nous intéresse ici, c'est-à-dire au niveau architectural. Il en est de même pour les modèles proches de la programmation tels que Java/A [BHH+06] ou ArchJava [ACN02b]. Certains modèles [BCL+06, CFP+03] proposent de repousser le langage de données au niveau de l'implantation : les types de données et les calculs associés sont alors définis, implantés et vérifiés par le compilateur. Ceci pose problème car les aspects fonctionnels du système ne sont pas pris en compte dès la phase de conception. De plus, ces derniers ne sont souvent pas intégrés avec les autres aspects (structurels et dynamiques, *etc.*). D'autres modèles [CR05, Sch03] prennent en compte des types de données et des contrats mais pas d'aspects dynamiques.

Les modèles avancés sur les aspects dynamiques [CFP+03, YS97, MKG99b, BCFS05, All97, BHP06] supportent des techniques poussées pour vérifier la correction des interactions mais manquent d'expressivité sur les types de données et ne fournissent pas de mécanismes d'assertion et donc de vérification associée. Dans Wright par exemple, la partie comportementale basée sur CSP est très détaillée (spécification et vérification) tandis que la partie liée aux données est minime « *We will not carry out any specific formal proof using the developed model.* » [All97] . Cependant, un modèle d'état et des opérations y sont décrits dans un sous-ensemble de Z. Dans [PNPR05], la prise en compte des aspects liés aux données se fait sous

forme de spécifications algébriques qui s'intègrent dans une vérification symbolique avec des systèmes de transition. Toutefois, le modèle ne supporte pas d'assertions.

Certaines approches se focalisent sur les contrats dans les communications. Par exemple, dans [Car03], l'idée est de définir des abstractions de communications (sortes de collaborations à la UML) puis de les réifier par des *mediums* ou composants de communication. La partie contrat est décrite en OCL. Dans [CFN03b], l'idée est d'associer des contraintes (*may/must*) aux interactions définies dans les interfaces, et ainsi de définir des contrats comportementaux liant le client et le serveur. Par ailleurs, un service est atomique dans le modèle de Carrez (une opération avec son contrat), alors que dans Kmelia il est hiérarchique et son comportement dynamique (eLTS) doit respecter les assertions.

Fractal [CS06] propose différentes approches basées sur la séparation des préoccupations. L'aspect structurel est pris en compte dans Fractal ADL [CQSS07] ; les assertions sont traitées dans ConFract [CR05] et enfin la dynamique est étudiée dans Vercors [BCMR07] ou Fractal/SOFA [BDH+08].

Nous pouvons donc constater que les modèles de composants actuels n'intègrent pas les différents aspects de modélisation dans un cadre cohérent et outillé. Même s'il existe des approches de spécification formelle dans le domaine du CBSE telles que [BHP06, MKG99a, All97, CRCR05], les aspects fonctionnels ne sont souvent pas considérés au premier plan mais sont plutôt intégrés lors des phases ultérieures du développement (par exemple, en utilisant les techniques de programmation défensive (cf. section 2.3.6).

Cette approche ne nous semble pas intéressante et en particulier dans le cas de systèmes critiques. L'intérêt des méthodes formelles est la possibilité de vérifier des propriétés sur le modèle. Si les aspects fonctionnels ne sont pas pris en compte dès les premières phases du développement, le système risque de ne pas correspondre aux attentes du client.

Pour combler ce manque, nous allons nous baser sur le modèle Kmelia présenté dans le chapitre précédent. Notre but est d'intégrer les aspects structurels, fonctionnels et dynamiques dans un même langage (Kmelia). Nous nous focalisons donc sur la définition d'un langage pour exprimer l'aspect fonctionnel. Ce langage devrait servir non seulement à la spécification et à la vérification des aspects fonctionnels, mais également à la vérification de la cohérence de ces derniers avec les autres aspects du système.

4.1.2 Problématique

Les travaux antérieurs autour de Kmelia portaient sur la définition des concepts structurels (composant, interface, service, composition, *etc.*) et la vérification de la partie comportementale de la composition de composants [AAA06c, AAA08]. Dans cette thèse, nous nous intéressons plus particulièrement à la modélisation et à la vérification des aspects fonctionnels qui n'ont pas été étudiées en profondeur (par exemple les expressions et les assertions étaient considérées comme de simples chaînes de caractères dépourvues de toute sémantique).

Nous avons donc besoin d'un langage de données pour décrire l'aspect fonctionnel

de Kmelia qui couvre la définition des types de données, l'état des composants, les expressions, les assertions des services (pré/post-conditions), les communications, les gardes des transitions, le typage des composants et des assemblages. Ainsi, ce langage représentera le support de base pour l'intégration de contrats permettant d'exprimer et de vérifier la correction des systèmes à composants modélisés en Kmelia.

En revanche, la prise en compte de l'aspect fonctionnel dans Kmelia entraîne de nouvelles caractéristiques dans le modèle qui, parfois, nécessitent la révision des autres aspects (structurels comme l'observabilité et dynamique comme la prise en compte des données et des gardes dans les actions). Nous détaillons ce point dans la section 4.4.

Comme nous l'avons expliqué dans le chapitre d'introduction, l'utilisation des méthodes de spécification formelle permet de vérifier tôt dans le cycle de développement des systèmes afin de répondre aux exigences des clients. L'aspect fonctionnel des systèmes à base de composants a donc tout intérêt à être modélisé de manière formelle.

La plupart des approches formelles utilisées aujourd'hui se focalisent sur une seule facette du développement des systèmes. Par exemple, une approche basée sur les états, comme Z [Spi89] ou B [Abr96], permet de bien caractériser les structures de données et les effets des opérations sur les états, ainsi que les propriétés d'invariance sur les transitions d'états, tandis qu'une approche basée sur les événements, comme CSP [Hoa83], CCS [Mil82], met en avant les comportements possibles d'un système, comme les propriétés de vivacité ou d'ordonnancement.

Comme un unique langage formel, s'il existait, permettrait difficilement de prendre en compte à la fois les aspects fonctionnels et les aspects dynamiques, nous recourons à l'intégration des méthodes formelles adaptées à chacune des facettes identifiées (fonctionnelle et comportementale). Nous pensons que deux langages pourraient être complémentaires pour résoudre ce problème. La complémentarité des informations et des propriétés modélisées par ces deux types d'approches permet de spécifier et de vérifier au mieux les systèmes à base de composants. De plus, l'isolation des problèmes permet de mieux les appréhender et les caractériser.

4.1.3 Besoins et alternatives

L'aspect fonctionnel devrait fournir une sémantique pour les éléments manquants dans Kmelia (types de données, expressions et contrats) et prendre en compte la sémantique des actions dans le comportement des services Kmelia. Nous avons donc besoin de moyens pour exprimer :

- l'espace d'états des composants : variables, constantes, types, *etc.*,
- les expressions logiques : assertions (invariant, pré/post-conditions), propriétés et gardes,
- la portée et l'observabilité de l'espace d'états des composants,
- les actions (requête, calcul, modification),
- le raisonnement sur le système et les propriétés définies par l'utilisateur.

Les besoins ci-dessus peuvent être satisfaits par un (ou plusieurs) langage(s). Un tel langage devrait également répondre à d'autres exigences telles que l'expressivité,

la compatibilité, l'extensibilité, *etc.*

Les objectifs ci-dessus entraînent des compromis dans le choix de notre solution, à savoir ; doit-on utiliser un seul langage couvrant tous les besoins ou plusieurs langages, chacun dédié à un besoin spécifique ? Doit-on choisir un langage existant ou un nouveau langage ?

Un ou plusieurs langages ? Nous pensons qu'un langage commun serait plus pratique (plus cohérent, plus simple). Mais s'il ne peut pas couvrir l'ensemble des besoins, nous pouvons étudier l'intégration de plusieurs langages. Par exemple, les types de données peuvent être définis en utilisant l'un des langages de spécification algébrique [Rua84] ; la définition de l'état et des assertions pourrait être spécifiée en utilisant une approche basée état (Z [Spi89] et B [Abr96])

Langage existant ou nouveau langage ? Il serait plus simple d'utiliser les langages et les outils existants. Mais s'ils ne peuvent pas couvrir toutes les caractéristiques attendues, on peut définir un nouveau langage. En revanche, afin de réutiliser les outils existants, on peut définir un algorithme de traduction vers le langage d'entrée de l'outil-cible. A titre d'exemple, OCL (*Object Constraint Language*) est un langage expressif et déclaratif qui peut être en partie traduit en Z ou B pour vérifier des propriétés.

4.1.4 Vers un langage de données intégré

À première vue, une première solution consisterait à utiliser un sous-ensemble du langage OCL restreint pour les expressions (sans navigation ni classes dans un premier temps) car il est expressif et les outils qui lui sont associés sont disponibles. Une sémantique simple pour les actions élémentaires peut être utilisée avec des expressions OCL et une affectation. On peut alors ensuite effectuer une traduction en B, CASL ou Maude pour prouver les propriétés exprimées en OCL. Une autre solution consisterait à étendre la grammaire d'expressions B. Une troisième solution consisterait à réutiliser une grammaire d'expressions existante qui supporterait les prédicats et les types de données de base. Ceux-ci peuvent facilement être traduits dans le langage de preuve.

Dans les solutions susmentionnées, l'intégration des expressions dans la grammaire Kmelia devrait être étudiée. Afin d'assurer une intégration harmonieuse, nous décidons alors de définir notre propre langage d'expressions et d'actions. L'avantage qui en résulte est l'abstraction et l'extensibilité du langage. L'inconvénient réside dans la difficulté de compilation d'un tel langage. Dans la suite, nous présentons en détail les éléments de base du langage de données que nous avons intégré dans la grammaire du modèle Kmelia.

4.2 Langage de données associé au modèle Kmelia

4.2.1 Éléments lexicographiques et syntaxiques

Dans Kmelia, les identificateurs sont composés de lettres, de chiffres et du caractère "_" suivant les règles usuelles. Dans ce qui suit, chaque classe d'identificateurs

est dénotée par un symbole non-terminal défini comme suit : C pour les constantes, V pour les variables, O pour les opérateurs et T pour les types.

⟨*letter*⟩	: :=	a..z | A..Z
⟨*under*⟩	: :=	_
⟨*number*⟩	: :=	⟨*digit*⟩ (⟨*digit*⟩)*
⟨*digit*⟩	: :=	0..9
⟨*word*⟩	: :=	⟨*letter*⟩ (⟨*digit*⟩ | ⟨*letter*⟩ | ⟨*under*⟩)*

4.2.2 Les types de données

Le modèle Kmelia permet de définir et de manipuler des structures de données abstraites [Rua84], dont la sémantique est largement inspirée des langages B [AH07a] et Z [Spi89]. Dans une large mesure, le choix des types abstraits pour la description des structures de données est conforme aux objectifs d'un langage de spécification, à savoir un modèle mathématique exprimant les propriétés que doit vérifier toute réalisation, sans imposer de contraintes d'implantation superflues.

Les données manipulées par Kmelia sont nombreuses et lui confèrent une expressivité importante. Ainsi nous avons repris les types de base usuels Integer, Boolean, Char, String, et nous avons fourni à l'utilisateur un moyen de définir ses propres types (tels que les énumérations, les tableaux, les ensembles, les structures, *etc.*) en suivant les règles ci-dessous :

⟨*integer_type*⟩	**: :=**	**INT**
⟨*bool_type*⟩	**: :=**	**BOOL**
⟨*string_type*⟩	**: :=**	**STRING**
⟨*typedef*⟩	**: :=**	(⟨*typename*⟩ | ⟨*struct*⟩ | ⟨*array*⟩ | ⟨*enum*⟩ | ⟨*range*⟩| ⟨*list*⟩ | ⟨*set*⟩)
⟨*struct*⟩	**: :=**	**struct {** ⟨*decl_vars*⟩ **}**
⟨*array*⟩	**: :=**	**array [range** | ⟨*typename*⟩ **] of** ⟨*typename*⟩
⟨*enum*⟩	**: :=**	**enum {** ⟨*word*⟩ **(,** ⟨*word*⟩ **)*}**
⟨*range*⟩	**: :=**	**range (** ⟨*number*⟩ | ⟨*identPrimary*⟩ **)** ⟨*period*⟩ ⟨*period*⟩ **(** ⟨*number*⟩ | ⟨*identPrimary*⟩ **)**
⟨*set*⟩	**: :=**	**setOf** ⟨*typename*⟩
⟨*list*⟩	**: :=**	**listOf** ⟨*typename*⟩
⟨*typename*⟩	**: :=**	⟨*word*⟩
⟨*decl_vars*⟩	**: :=**	⟨*dec_v_untyp*⟩ **(;** ⟨*dec_v_untyp*⟩ **)***
⟨*dec_v_untyp*⟩	**: :=**	**(obs) ? (**⟨*word*⟩ **(,** ⟨*word*⟩ **)*** **)+ :** ⟨*typedef*⟩

L'utilisateur peut déclarer ses propres types dans les spécifications de composants (dans la clause **TYPES**) ou importer des types prédéfinis dans des bibliothèques en utilisant la clause **USES**, par exemple, **USES** {STOCKLIB} (cf. l'exemple de la section 4.3). Une bibliothèque Kmelia fournit une liste de types, de constantes et de signatures de fonctions. La définition de Kmelia n'impose aucune contrainte sur la façon dont cette bibliothèque doit être réalisée.

4.2.3 Les expressions

Une expression Kmelia est construite avec des constantes C, des variables V et des opérateurs arithmétiques et logiques usuels O $(+, *, mod, <, >=, etc.)$.

⟨*primaryExp*⟩	: :=	⟨*identPrimary*⟩ | ⟨*constant*⟩ | **true** | **false** | (⟨*condExp*⟩)
⟨*identPrimary*⟩	: :=	⟨*word*⟩ (⟨*period*⟩ ⟨*word*⟩)*
⟨*exp*⟩	: :=	⟨*assignExp*⟩
⟨*expList*⟩	: :=	⟨*exp*⟩ (, ⟨*exp*⟩)*
⟨*assignExp*⟩	: :=	⟨*condExp*⟩ (:= (⟨*combis*⟩ | ⟨*condExp*⟩)) ?
⟨*condExp*⟩	: :=	⟨*logicImpliesExp*⟩ | ⟨*forallpred*⟩ | ⟨*existspred*⟩ | ⟨*unchangedpred*⟩
⟨*logicImpliesExp*⟩	: :=	⟨*logicEquivExp*⟩ (**implies** ⟨*condExp*⟩)*
⟨*logicEquivExp*⟩	: :=	⟨*logicXorExp*⟩ (**equiv** ⟨*condExp*⟩)*
⟨*logicXorExp*⟩	: :=	⟨*logicOrExp*⟩ (**xor** ⟨*condExp*⟩)*
⟨*logicOrExp*⟩	: :=	⟨*logicAndExp*⟩ (**or** ⟨*condExp*⟩)*
⟨*logicAndExp*⟩	: :=	⟨*equalityExp*⟩ (**and** ⟨*condExp*⟩)*
⟨*eqExp*⟩	: :=	⟨*relExp*⟩ ((<> | =) ⟨*relExp*⟩)*
⟨*relExp*⟩	: :=	⟨*addExp*⟩ ((< | > | <= | >= | = | **in**)⟨*addExp*⟩)*
⟨*addExp*⟩	: :=	⟨*multExp*⟩ ((+ | -) ⟨*multExp*⟩)*
⟨*multExp*⟩	: :=	⟨*unaryExp*⟩ ((* | /) ⟨*unaryExp*⟩)*
⟨*unaryExp*⟩	: :=	- ⟨*unaryExp*⟩ | + ⟨*unaryExp*⟩ | | **not** ⟨*unaryExp*⟩

4.2.4 Les prédicats, assertions et propriétés

Nous avons introduit des assertions sous forme de prédicats (pré/post-conditions, invariants, gardes, propriétés, ...). Voici la syntaxe simplifiée des prédicats :

⟨*properties*⟩	: :=	⟨*property*⟩ (, ⟨*property*⟩)*
⟨*property*⟩	: :=	(**obs** | **local**) ? (@ ⟨*word*⟩) ? (:) ? ⟨*pred*⟩
⟨*pred*⟩	: :=	(⟨*forallpred*⟩) ? (⟨*existspred*⟩) ? (|) ? ⟨*exp*⟩
⟨*forallpred*⟩	: :=	**forall** ⟨*decl_vars*⟩ (**exists** ⟨*decl_vars*⟩) ? | ⟨*condExp*⟩
⟨*existspred*⟩	: :=	**exists** ⟨*decl_vars*⟩ | ⟨*condExp*⟩
⟨*unchangedpred*⟩	: :=	**Unchanged** ⟨*identPrimary*⟩ (, ⟨*identPrimary*⟩)*

Un **invariant** de composant est une propriété qui doit être vraie à tout instant et qui est vérifiée avant et après l'exécution de chaque service.

Une **pré-condition** de service est un prédicat. Elle porte sur les arguments en entrée que le client (appelant) doit respecter. Si la pré-condition n'est pas respectée, le fournisseur (appelé) ne s'engage pas à exécuter correctement le service appelé. Les pré-conditions doivent être nécessaires et suffisantes pour que l'appelant puisse être servi et obtenir les garanties associées aux post-conditions.

Une **post-condition** est un prédicat qui garantit les sorties que le fournisseur (appelé) s'engage à respecter si le service s'exécute normalement. Le mot-clef old peut être utilisé dans les post-conditions pour faire référence à l'état dans lequel étaient les variables du composant juste avant l'exécution du service.

⟨*compinvariant*⟩	: :=	**INVARIANT** ⟨*properties*⟩
⟨*compproperties*⟩	: :=	**PROPERTIES** ⟨*properties*⟩
⟨*compservproperties*⟩	: :=	**SERVICE_PROPERTIES** ⟨*properties*⟩
⟨*preclause*⟩	: :=	**Pre** ⟨*properties*⟩
⟨*postclause*⟩	: :=	**Post** ⟨*properties*⟩

4.3 Étude de cas : un système de gestion de stock

Dans le cadre de la description du langage de données et de la vérification de contrats (cf. chapitres suivants), nous nous appuierons sur des exemples issus d'un système de composants modélisant une application de gestion de stock, conçu uniquement à des fins d'illustration [MAA10b]. Notons que cette application est une modélisation simplifiée du système Inventory de la figure 3.6 du cas CoCoME.

Il s'agit d'un système modélisant un magasin qui tient des dépôts de différents types d'articles, stockés par une société. Le système enregistre le niveau de stock pour chaque type d'article stocké, l'entrée en stock et la sortie du stock des articles. Parfois un nouveau type d'article est référencé (article qui sera stocké) et il arrive aussi que des types d'articles soient abandonnés (dé-référencés) quand leur niveau de stock est nul. Chaque article est caractérisé par une référence unique et la quantité disponible en stock. Le système permet donc de gérer les références dans un catalogue (catalog) et les quantités stockées (stock).

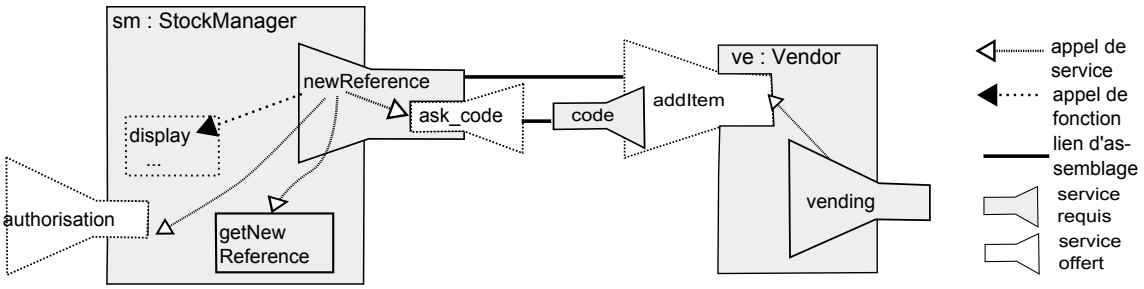


FIGURE 4.1 – Assemblage Kmelia simplifié d'une application de gestion de stock

Le magasinier (le composant Vendor) peut ajouter ou retirer des références selon les règles de gestion établies telles que : *on ne peut ajouter une référence que si elle ne figure pas dans le catalogue et on ne peut retirer une référence que si son stock est vide.*

La figure 4.1 illustre une vue partielle d'un **assemblage** de ce système. Les "entonnoirs" gris (resp. blancs) désignent des services offerts (resp. requis). Cet assemblage est décrit en Kmelia dans le listing 4.1.

Listing 4.1 – Spécification Kmelia d'un assemblage de deux composants

```
Assembly
  Components
    sm : StockManager;
    ve : Vendor
  Links ////////////assembly links/////////
    lref: p-r sm.newReference, ve.addItem
          context mapping
                  ve.catalogEmpty == empty(sm.catalog),
                  ve.catalogFull == size(sm.catalog) = MaxInt
  sublinks : {lcode}
    lcode: r-p sm.ask_code, ve.code
   ...
End // assembly
```

Les **correspondances de contexte et de message** (cf. section 4.4.2) sont spécifiées dans les **liens d'assemblage** (par exemple le lien lref). Dans le listing 4.1, les variables du contexte virtuel du service requis addItem sont associées à une expression définie sur les variables du contexte du service offert newReference, c'est-à-dire les variables d'état observables du composant sm:StockManager. Dans cet exemple, il n'existe pas de correspondance de message car les deux services utilisent le message msg (déclaré dans la bibliothèque par défaut de Kmelia).

Le listing 4.2 montre la spécification Kmelia du **composite** StockSystem de la représentation de la figure 4.2. Le composite StockSystem est défini par un assemblage de deux composants (sm:StockManager) et (ve:Vendor). Le premier est le composant de base permettant de gérer les références et le stock. Le second représente l'interface du système.

Listing 4.2 – Spécification Kmelia du composite StockSystem

```
COMPONENT StockSystem
INTERFACE
  provides : {vending}
  requires : {authorisation}
COMPOSITION
Assembly
  ...
End // assembly
Promotion
  Links ////////////promotion links/////////
    lvend: p-p ve.vending, SELF.vending
    laut: r-r sm.authorisation, SELF.authorisation
END_COMPOSITION
```

Les **liens de promotions** (par exemple les liens lvend et laut) rendent les services vending et authorisation disponibles au niveau du composite. Le service (vending) est donc **promu** au niveau du composite StockSystem. Le service vending modélise le processus de vente. Ce service requiert quatre services, dont le **service requis** addItem qui est lié au **service offert** newReference par un lien d'assemblage. Dans la suite, on se focalise sur les services addItem et newReference.

Le service vending permet d'ajouter un nouvel article dans le système en utilisant le service requis addItem. Ce dernier est satisfait par le service offert newReference qui fournit une nouvelle référence et effectue la mise à jour du système (cf. listing 4.4).

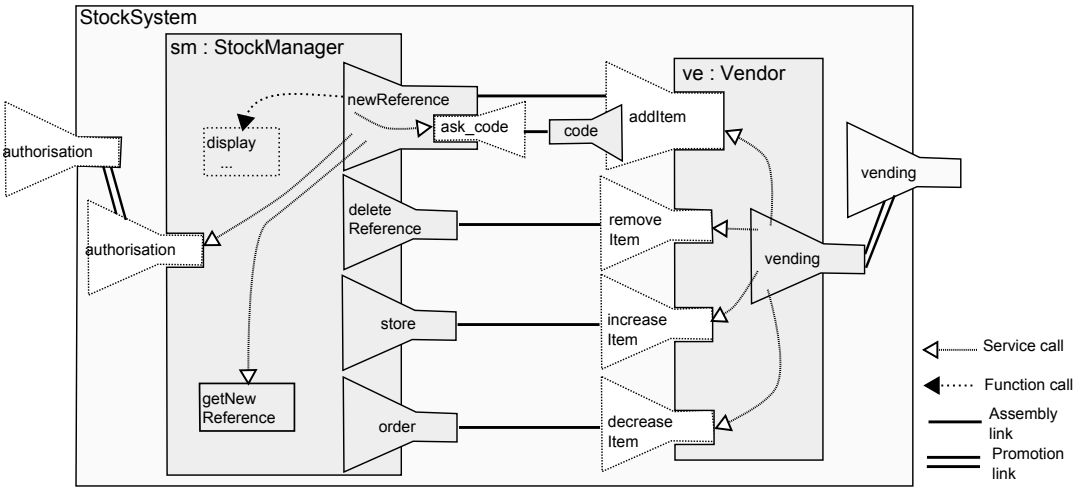


FIGURE 4.2 – Le composite StockSystem modélisant l'application de gestion de stock

Les données, les assertions et les expressions sont décrites en utilisant le langage de données introduit dans la section précédente. Les types de données sont explicitement définis dans la clause **TYPES** ou dans une **bibliothèque** partagée (prédéfinie ou définie par le spécifieur). À titre d'exemple, la bibliothèque ci-dessous (appelée StockLib) déclare certains types, fonctions et constantes.

```
TYPES
    ProductItem :: struct {id: Integer; desc: String; quantity: Integer} ;
CONSTANTS
    maxRef : Integer := 100;
    emptyString : String := "" ;
    noReference : Integer := -1;
    noQuantity : Integer := -1
```

Le listing 4.3 est extrait de la spécification Kmelia du **composant** StockManager. L'**espace d'état** du composant StockManager déclare une **variable** ***observable*** catalog et deux tableaux (plabels et pstock) sont utilisés pour stocker les étiquettes de références et les quantités disponibles.

Listing 4.3 – Spécification Kmelia de l'espace d'état du composant StockManager

```
COMPONENT StockManager
INTERFACE
  provides : {newReference, removeReference, storeItem, orderItem}
  requires : {authorisation}
USES {STOCKLIB} // library of the project types and functions
TYPES
  Reference :: range 1..maxRef
VARIABLES
  vendorCodes : setOf Integer; //authorised administrators
  obs catalog : setOf Reference; // product id = index of the arrays
  plabels : array [Reference] of String; //product description
  pstock : array [Reference] of Integer //product quantity
INVARIANT
INITIALIZATION
  catalog := emptySet;
  vendorCodes := emptySet;     //filled by a required service
  plabels:= arrayInit(plabels,emptyString); //consistent with ..
  pstock := arrayInit(pstock,noQuantity);  //..empty catalog
```

L'**invariant** du composant StockManager indique que le catalogue a une limite supérieure ; toutes les références dans le catalogue ont une étiquette et une quantité ; les références inconnues n'ont ni étiquette ni quantité.

```
INVARIANT
  obs @borned: size(catalog) <= maxRef,
  @referenced: forall ref : Reference | includes(catalog,ref) implies
     (plabels[ref] <> emptyString and pstock[ref] <> noQuantity),
  @notreferenced: forall ref : Reference | excludes(catalog,ref) implies
     (plabels[ref] = emptyString and pstock[ref] = noQuantity)
```

Notons que les prédicats peuvent être **nommés** (par exemple @borned dans le listing 4.3) afin d'alléger les spécifications et de faciliter la traçabilité.

Dans le listing 4.4, le service newReference vérifie le code d'accès du vendeur, puis lit et initialise les nouvelles références.

Listing 4.4 – Spécification Kmelia du service newReference

```
provided newReference () : Integer  //Result = ProductId or noReference
Interface
   calrequires : {ask_code} #required from the caller
   intrequires : {getNewReference}
Pre
   size(catalog) < maxRef #the catalog is not full
// LTS ci-dessous
Post
 @resultRange:((Result >=1 and Result <= maxRef) or (Result = noReference)),
 @resultValue:(Result <> noReference) implies (notIn(old(catalog),Result)
                 and catalog = add(old(catalog),Result)),
 @noresultValue:(Result = noReference) implies Unchanged{catalog},
 local @refAndQuantity: (Result <> noReference) implies
         (pstock[Result] = 0 and plabels[Result] <> emptyString and
         (forall i : Reference | (i <> Result) implies
            (pstock[i] = old(pstock)[i] and plabels[i] = old(plabels)[i] ))
         ),
 local @NorefAndQuantity: (Result = noReference) implies Unchanged{pstock,plabels}
End
```

La **pré-condition** indique que le catalogue ne doit pas être plein. Elle porte sur la **partie observable** de l'espace d'état du composant.

```
Pre
   size(catalog) < maxRef #the catalog is not full
```

La **post-condition** comprend une partie observable : le résultat (variable result) est soit une référence nulle (noReference) soit une nouvelle référence (dans l'intervalle autorisé) ; s'il s'agit d'une nouvelle référence, alors le nouveau catalogue est l'ancien catalogue (mot-clé old) auquel on a ajouté cette référence. Dans le cas d'une référence nulle, le catalogue est **inchangé** (mot-clé *unchanged*).

```
Post
 @resultRange:((Result >=1 and Result <= maxRef) or (Result = noReference)),
 @resultValue:(Result <> noReference) implies (notIn(old(catalog),Result)
                 and catalog = add(old(catalog),Result)),
 @noresultValue:(Result = noReference) implies Unchanged{catalog},
```

La post-condition comprend aussi une **partie non observable** (ou *locale*) indiquant les effets sur l'espace d'état non-observable.

```
local @refAndQuantity: (Result <> noReference) implies
        (pstock[Result] = 0 and plabels[Result] <> emptyString and
        (forall i : Reference | (i <> Result) implies
           (pstock[i] = old(pstock)[i] and plabels[i] = old(plabels)[i] ))
        ),
 local @NorefAndQuantity: (Result = noReference) implies Unchanged{pstock,plabels}
```

Dans le listing 4.4, le service newReference requiert un service ask_code de son appelant et un service interne getNewReference. Inversement, dans le listing 4.5 le service addItem propose un service code.

Pour un service requis, on "imagine" une partie de cet espace d'état observable, qu'on modélise sous forme d'un **espace d'état virtuel** (cf. section 4.4.2).

4.4 Enrichissement des contrats dans Kmelia

La prise en compte de données dans les composants relève des questions spécifiques lorsqu'on assemble et encapsule les composants, par exemple la portée des variables dans une cascade d'encapsulation, leur portée par rapport aux assertions, leur correspondance dans les assemblages, *etc.*

4.4.1 Portée et règles d'observabilité

Dans le but de permettre la conception et la composition des composants indépendamment des contextes précis d'utilisation, nous avons introduit dans Kmelia une notion d'**observabilité** de l'état des composants. Elle est assurée en utilisant soit des variables observables soit l'accessibilité des méthodes ou encore des contrôleurs d'attributs.

En complément de l'interface d'un composant Kmelia, son état peut être observé par des services, par des composants qui l'utiliseraient ou par des composites qui l'encapsuleraient. Par conséquent, l'ensemble des variables d'état V d'un composant est partitionné en deux sous-ensembles, l'un contenant les variables observables V^O et l'autre contenant les variables non-observables V^{NO}. De même, un prédicat *pred* (par exemple, dans un invariant, une pré ou une post-condition) peut être rendu observable et sera noté dans la suite $pred^O$. L'observabilité dans Kmelia est exprimée par le mot-clé **obs**. Par exemple, dans le listing 4.3, seule la variable d'état catalog qui contient les références de produits est **observable**. Les codes vendeurs, les libellés et les quantités en stock ne le sont pas.

```
obs catalog : setOf Reference ; // product_id = index of the arrays
```

On appelle **espace d'état observable** le sous-ensemble qui est déclaré visible par le concepteur du composant. En pratique, on utilise l'espace d'état observable pour définir des contrats de composition (assemblage ou promotion).

Les pré/post-conditions de s et le contexte virtuel doivent respecter les règles d'observabilité suivantes :

- Pour un service fourni, la pré-condition ne doit pas contenir de prédicats non-observables : le client n'aurait dans ce cas aucun moyen de savoir si la pré-condition est satisfaite,

– à l'inverse, pour un service requis, c'est la post-condition qui exprime ce qui est attendu, elle ne doit donc pas contenir de prédicats non-observables.

Portée des assertions	Pré-condition		Post-condition	
	Observable	Non-observable	Observable	Non-observable
du service	Pre^{O}	Pre^{NO}	$Post^{O}$	$Post^{NO}$
`Offert` s	$V^{O} \cup param$	-	$V^{O} \cup param \cup \{$ result $\}$	$V \cup param \cup \{$ result $\}$
`Requis` s	$vV \cup param$	$V \cup param$	$vV \cup param \cup \{$ result $\}$	-

Dans le tableau ci-dessus, vV dénote les variables locales d'un service requis déclarées dans le contexte virtuel et *param* (resp. *result*) dénote les paramètres (resp. la variable de retour) d'un service (offert ou requis).

La figure 4.3 résume les relations entre l'espace d'état du composant et l'observabilité des variables et des assertions des services.

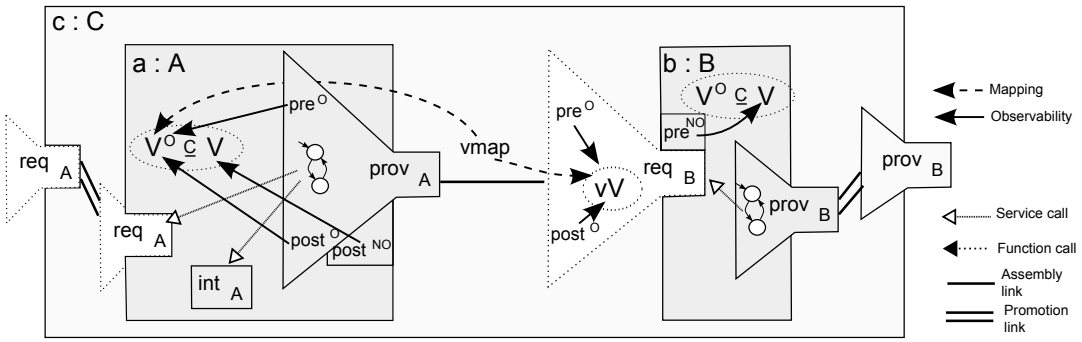


FIGURE 4.3 – La portée des variables d'état et des assertions

Les pré/post-conditions observables d'un service prov_A (resp. req_B) se réfèrent à l'état observable V^O de a (resp. au contexte virtuel vV de req_B). Ces conditions sont les bases pour vérifier les contrats d'assemblage et les contrats de promotions (cf. chapitre 6). Le contexte virtuel vV de req_B devrait correspondre à un sous-ensemble de variables définies comme observables dans le composant a.

Les pré-conditions (resp. les post-conditions) non-observables n'ont pas de sens pour un service offert (resp. pour un service requis), car elles permettent l'expression d'autres exigences qui ne seront pas forcément respectées par le client (resp. le fournisseur) du service.

L'abstraction et l'**encapsulation** sont cruciales pour le passage à l'échelle dans les approches à base de composants. Pour un observateur extérieur, un composite ne devrait pas être distingué d'un composant primitif et ne devrait pas être trop complexe. Promouvoir tous les services et les variables de ses sous-composants ne permettrait pas d'atteindre cet objectif. Au moment de décider ce qui devrait être observable ou ce qui devrait être promu, le spécifieur fait un compromis entre l'encapsulation et la description précise de l'état.

4.4.2 Enrichissement du contrat de clientèle

En plus de la séparation des fonctionnalités offertes et requises, les besoins d'un composant Kmelia, exprimés en terme de services requis, doivent être suffisamment

précis et complets pour exprimer un contrat de clientèle. Un tel contrat facilite, d'une part, la recherche de services pouvant satisfaire ce service requis et, d'autre part, la vérification en isolant les vérifications internes au composant des vérifications externes.

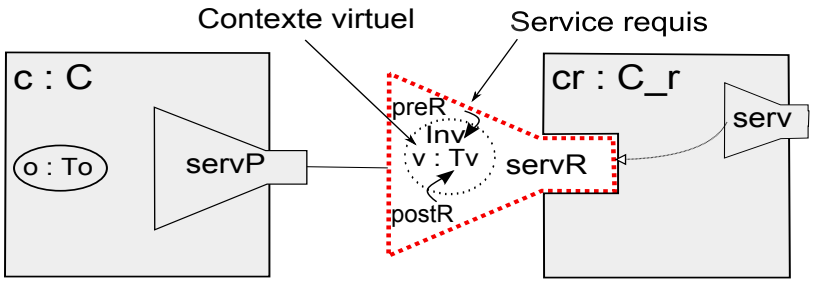


FIGURE 4.4 – Contexte virtuel d'un service requis

Un service requis *servR* d'un composant *C_r* est une abstraction d'un service offert par un autre composant *a priori* inconnu du composant *C_r*.

Le contexte virtuel : Pour pouvoir poser des hypothèses précises sur le fournisseur d'un service requis, nous avons introduit la notion d'*espace d'états virtuel* $\mathcal{W}^V$ afin d'abstraire un service de son contexte de définition qui est un composant. L'espace d'états virtuel caractérisant un composant C dans un service requis d'un autre composant *C_r* est décrit par des variables et un invariant qui sont supposés être compatibles avec les variables observables du composant C.

Listing 4.5 – Spécification Kmelia du service addItem

```
required addItem () : Integer
Interface
  subprovides : {code}
Virtual Variables
  catalogFull : Boolean;
  catalogEmpty : Boolean     //possibly catalogSize
Virtual Invariant not(catalogEmpty and catalogFull)
Pre
  True
Behavior  //No LTS
Post
  @ref: (Result <> noReference) implies (not catalogEmpty),
```

Dans l'exemple du listing 4.5, le service addItem perçoit le catalogue sous forme de deux booléens catalogFull et catalogEmpty. La pré-condition "observable" porte sur cet espace virtuel, de même que la post-condition. La pré-condition "locale" ajoute des conditions liées à l'espace d'état du composant du service requis.

Correspondance de contextes et de messages : Dans le cadre des liens d'assemblage, le spécifieur doit établir deux correspondances afin d'assurer la cohérence des liens et donc de l'assemblage global. La première correspondance établit explicitement la relation entre le contexte virtuel du service requis et le contexte observable du composant du service offert (*context mapping*). Elle illustre bien la séparation des

besoins de leur satisfaction : le service requis est défini sur un ensemble d'hypothèses (un espace virtuel) qu'on confronte avec un composant concret (des contraintes réelles).

Soit un service requis sr d'un composant CR, lié à un service offert sp d'un composant CP par un lien d'assemblage. Les variables de l'espace d'états virtuel (vV_{sr}) de sr sont mises en correspondance avec les variables *observables* de CP (V^O_{CP}) par une fonction totale $vmap : vV_{sr} \rightarrow exp(V^O_{CP})$ où $exp(V)$ désigne une expression sur les variables de V.

Définition 4.1 (Correspondance de contexte) *La correspondance de contexte est une fonction de concrétisation associée à chaque lien d'assemblage et définie comme suit :*

$$
\begin{array}{ll}
& vmapVar : BaseLink \leftrightarrow V \\
& vmapExp : (BaseLink \times V) \nrightarrow exp(V) \\
(1) & \operatorname{dom} vmapVar \subseteq (alinks \cup subs) \land \operatorname{dom} vmapExp = vmapVar \land \\
& \quad (\forall (c_i, n_1, c_j, n_2) : BaseLink \mid (c_i, n_1, c_j, n_2) \in \operatorname{dom} vmapVar \bullet \\
(2) & \qquad vmapVar(\!|\{(c_i, n_1, c_j, n_2)\}|\!) = V^V_{\nu(n_2)} \land \\
(3) & \qquad (\forall v : V^V_{\nu(n_2)} \bullet var(vmapExp((c_i, n_1, c_j, n_2), v)) \subseteq V^O_{C_i}))
\end{array}
$$

où $exp(V)$ désigne l'ensemble des expressions sur l'ensemble V. Les contraintes expriment le fait que la correspondance se fait sur les liens de l'assemblage considéré (1), que toutes les variables du service requis n_2 ont une expression (2) constituée à partir des variables observables du composant de n_1 (3).

Par exemple, la correspondance de contexte définie sur le lien d'assemblage lref du listing suivant permet d'exprimer que la variable catalogue du composant StockManager est perçue comme deux variables booléennes par le service requis addItem du composant Vendor. La variable catalogEmpty (resp. catalogFull) modélise le fait que le catalogue soit vide (resp. plein). Chacune des deux variables est calculée par une expression définie sur des variables concrètes du composant fournissant le service.

```
Links ////////////assembly links/////////
  lref: p-r sm.newReference, ve.addItem
       context mapping
               ve.catalogEmpty == empty(sm.catalog),
               ve.catalogFull == size(sm.catalog) = MaxInt
  //...
```

La seconde correspondance permet de relier explicitement les identifiants des messages utilisés dans les services liés : c'est la correspondance de messages (*message mapping*). Pour un lien donné, chaque nom de message de sr est associé à un nom de message de sp par une bijection totale $mmap : mname_{sr} \rightarrowtail\!\!\!\!\!\rightarrow mname_{sp}$. Où $mname_{sr}$ (resp. $mname_{sp}$) est l'ensemble de messages définis dans le service sr (resp. le service sp).

4.4.3 Métamodèle des contrats

La figure 4.5 présente le métamodèle Kmelia enrichi par les contrats [AAM11]. Nous rappelons que les composants Kmelia sont caractérisés par un état et des services. Ils sont assemblés sur leurs points d'accès (*EndPoint*) qui sont des interfaces

composées d'une liste de services. Un assemblage relie les composants, éventuellement via d'autres assemblages, sur une relation de clientèle (contrat client/serveur classique). Un composite les encapsule dans une relation d'inclusion (contrat parent/enfant). Nous remarquons que toute les liaisons entre composant se font au niveau des services.

Chaque élément de modélisation définit un contrat. Un contrat peut être, lui-même, composite. Par exemple le contrat de clientèle défini au niveau d'un assemblage est composé d'un contrat structurel (signatures), d'un contrat fonctionnel (pré/post-conditions) et d'un contrat comportemental (eLTS). Un contrat peut également contenir plusieurs éléments (ContractElement). Par exemple un contrat d'un service requis, contient un invariant, une pré-condition et une post-condition.

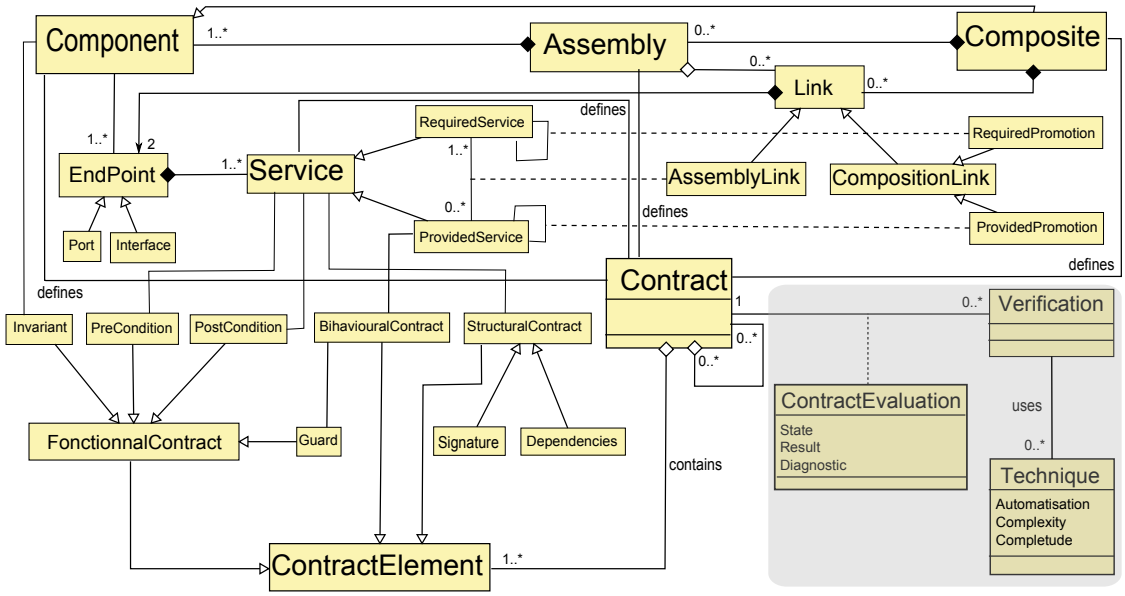


FIGURE 4.5 – Métamodèle simplifié du modèle Kmelia enrichi par les contrats

La partie grisée montre que plusieurs techniques peuvent être utilisées pour vérifier un même contrat élémentaire. Les techniques de vérification de contrats sont caractérisées par leur degré d'automatisation, leur complexité (qui peut être calculée en fonction du modèle) et la force du résultat (une preuve de propriété est plus forte que l'absence d'échecs lors d'un test, par exemple). Une vérification effectuée produit un résultat, partiel ou complet, assorti d'un diagnostic exploitable par d'autres vérifications.

4.5 Conclusion

Dans ce chapitre, nous avons présenté notre contribution à l'enrichissement du modèle Kmelia, à savoir un langage de données pour les expressions et les prédicats, une notion d'observabilité et des contrats de service. Ce langage de données servira tout d'abord dans l'étape de spécification du système puis dans l'étape de vérification. L'analyse statique est opérationnelle dans l'outil COSTO associé au

modèle Kmelia. Nous avons illustré cette démarche sur un exemple simplifié d'une application de gestion de stock.

L'intégration du langage de données fait de Kmelia un langage large-spectre qui :

- couvre à la fois les aspects structurels, fonctionnels et comportementaux ;
- renforce la notion de requis (par la notion du contexte virtuel) facilitant une réutilisation individuelle (composants *off-the-shelf*) ;
- renforce la notion d'observabilité en favorisant ainsi la construction compositionnelle.

La prise en compte de données dans Kmelia permet d'exprimer des propriétés de cohérence plus avancées au niveau des services (conformité du eLTS par rapport aux assertions pré/post-conditions), au niveau des composants (préservation de l'invariant), des assemblages (le contrat de clientèle entre un service offert et requis) et de composition (encapsulation et correction de la promotion). La vérification de telles propriétés doit s'intégrer aux vérifications de propriétés sur les aspects structurels et dynamiques. Dans les deux chapitres suivants, nous détaillons chacune de ces propriétés et définissons pour chacune d'elles les obligations à vérifier, ainsi que la technique de vérification associée.

Chapitre 5

Un cadre pour la spécification et la vérification de contrats

Dans ce chapitre, nous montrons comment tirer parti des contrats dans la développement des systèmes à composants. Nous présentons d'abord la notion de contrats multi-niveaux (section 5.1). Ensuite, nous proposons un processus de développement de composants basé sur l'utilisation de contrats (section 5.2). Enfin, nous nous intéressons à deux niveaux de vérification basés sur des obligations de preuve : d'une part, les obligations de preuve intra-composant associées aux composants et à leurs services (sections 5.3) et, d'autre part, les obligations de preuve inter-composants essentiellement liées aux assemblages et aux composites (sections 5.4).

5.1 Notion de contrats multi-niveaux

Comme nous l'avons évoqué dans les chapitres 2 et 3, la plupart des travaux proposent une vérification de type correction structurelle et comportementale d'une architecture. Notre objectif est d'enrichir la description des composants ainsi que leur composition afin de vérifier leur validité vis-à-vis des aspects fonctionnels. Selon [Mey03] : *" a Trusted Component is a reusable software element possessing specified and guaranteed property qualities "*. La notion de contrat est utile pour modéliser différents types de propriétés (comme nous l'avons déjà vu dans la section 2.3.7). Par exemple :

- ***les propriétés structurelles*** : la compatibilité des signatures d'interface (noms et types) ; est-ce qu'un composant donne assez d'informations sur son (ses) interface(s) afin d'être (ré)utilisable par d'autres composants ? La disponibilité des composants requis, la disponibilité des services requis et la correction des interfaces des composants liés ;
- ***les propriétés fonctionnelles*** : est-ce que les composants font ce qu'ils sont censés faire ? Ces propriétés peuvent être vérifiées sur les composants, les assemblages et les compositions ;
- ***les propriétés comportementales*** : l'interaction entre deux ou plusieurs composants assemblés est-elle correcte ? Les propriétés dépendent des caractéristiques du modèle d'interaction : séquentiel ou concurrent, synchrone ou

asynchrone, binaire ou n-aire avec partage de données, synchronisation, *etc.*

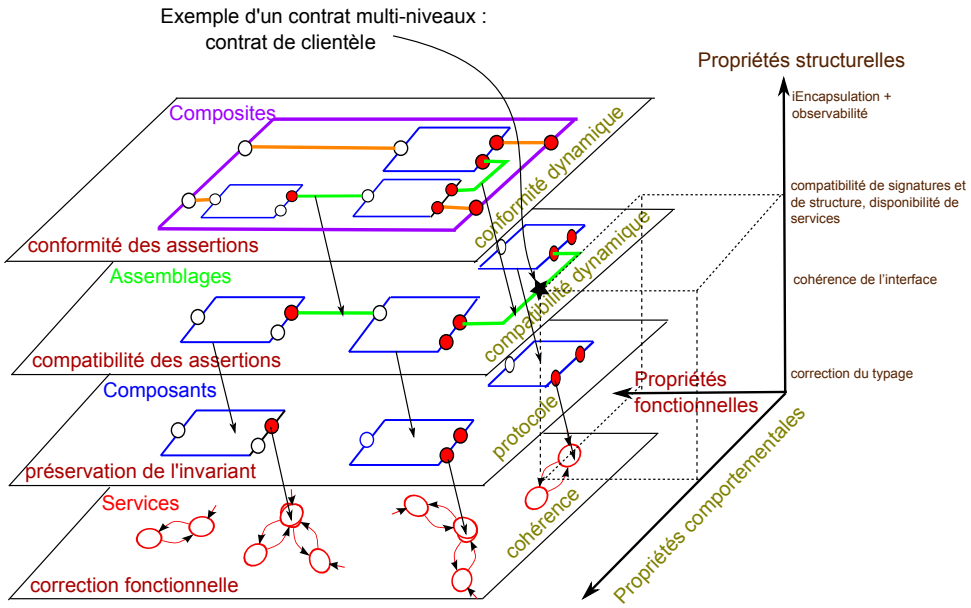


FIGURE 5.1 – Différents aspects de la modélisation des systèmes à base de composants.

Le **contrat de service** regroupe, en plus de la signature, la *cohérence comportementale* assurant que son exécution ne conduit pas à des états incohérents (tel que l'interblocage) d'une part, et d'autre part la *correction fonctionnelle* exprimant le fait qu'un service réalise ce qu'il est censé faire en utilisant l'axiomatique de Hoare. Le **contrat de composant** prévoit essentiellement l'accessibilité des services (chaîne de dépendance), la cohérence du composant (préservation de son invariant) ainsi que la correction du protocole (enchaînement licite des appels de services). Le **contrat d'assemblage** est un contrat de clientèle classique mais qui se décline sur trois niveaux de compatibilité. Enfin, le **contrat de composite** est une variante du précédent pour la relation parent/enfant (visibilité, promotion).

Afin de faire face aux différentes propriétés susmentionnées, nous introduisons la notion de contrat multi-niveaux (*multi-level contract*). La figure 5.1 illustre le fait que la correction est vérifiée à partir des éléments structurels, fonctionnels et comportementaux. Ces derniers peuvent se manifester sous forme de contrats définis au niveau des services comme des assertions (pré/post-conditions), au niveau des composants comme des invariants à préserver par les services, au niveau des assemblages pour garantir des propriétés de compatibilité de composants ou au niveau des composites pour s'assurer de la correction des services promus.

Définition 5.1 *Un contrat multi-niveaux est un contrat défini à différents niveaux de modélisation (service, composants, assemblage, composite) portant sur plusieurs aspects de la modélisation.*

Par exemple, le *contrat de clientèle* exprimé au niveau d'un assemblage se décline sur trois niveaux ; la compatibilité des signatures, la conformité des assertions ainsi

que la compatibilité des comportements. La vision hiérarchique des contrats fournit un cadre pratique pour maîtriser la construction progressive des systèmes à base de composants et le processus de leur vérification en séparant les préoccupations structurelles, fonctionnelles et dynamiques.

Dans la suite, nous présentons les apports des contrats (multi-niveaux) en terme de spécification et de vérification. Notons que les contrats peuvent également être utilisés à des fins de simulation et de test [1] mais, dans cette thèse, nous nous intéressons plus à l'utilisation des contrats dans la vérification des systèmes à composants.

5.2 Processus de développement des systèmes à base de composants et de contrats

Cette section présente un processus de développement à base de composants qui prend en compte les contrats multi-niveaux introduits dans la section 5.1. Les composants et les assemblages sont supposés être abstraits, ce qui signifie qu'ils sont indépendants des plates-formes d'exécution. Ils peuvent être raffinés ou mis en œuvre plus tard dans des plates-formes d'exécution.

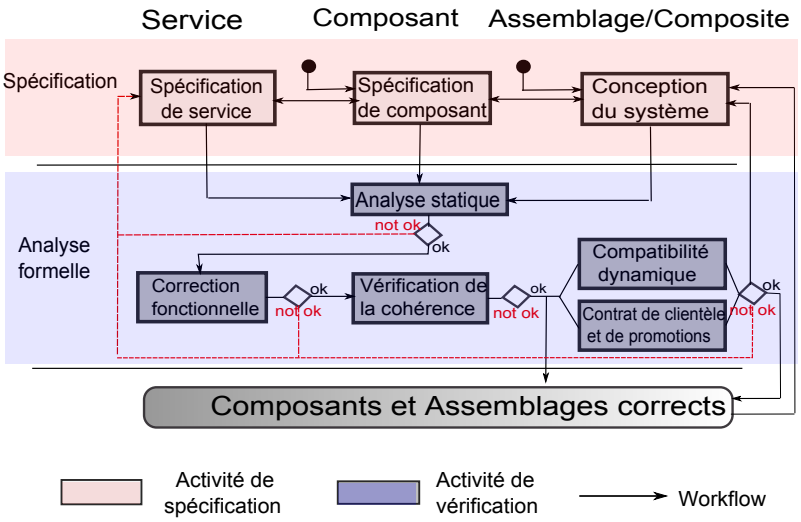


FIGURE 5.2 – Phases de développement de composants basé sur les contrats multi-niveaux

Comme le montre la figure 5.2, le processus est divisé en deux phases : la phase de spécification, faite d'activités de spécification et la phase d'analyse formelle, faite des activités de vérification. Les activités peuvent être effectuées de façon itérative. D'un point de vue pratique, le spécifieur peut passer d'une phase à l'autre en fonction de sa propre méthodologie, inspirée des approches ascendantes ou descendantes. Par exemple, le spécifieur peut itérer le niveau des composants uniquement pour fournir des composants sur étagère. Cette approche de conception permet la réutilisation

1. Un travail de thèse, dans l'équipe AeLoS, commencé par Olivier Finot consiste à utiliser les contrats pour le test des modèles (notamment les modèles de composants).

des composants conçus en faisant la description des composants disponibles dans une bibliothèque de composants.

5.2.1 Démarche de conception descendante *vs.* ascendante

En modélisation, on distingue essentiellement les approches *top-down* (descendante) et *bottom-up* (ascendante) comme méthodes avec lesquelles l'organisation hiérarchique interne d'un système complexe est décrite. Selon [Edm99], une modélisation *top-down* est une approche dans laquelle on tente de définir l'organisation d'un système en formulant d'abord les principes généraux pour aller ensuite vers le détail. Inversement, une modélisation *bottom-up* consiste à aborder la modélisation par les détails pour en produire les abstractions ou généralisations appropriées. En fonction du domaine et de la connaissance de la structure et de la fonction d'un système, il peut parfois être difficile d'établir des principes généraux.

La création de services peut intervenir dans une démarche de conception descendante ou ascendante. Dans le premier cas, les services sont des squelettes destinés à être enrichis ou raffinés. Dans le second cas, plusieurs services existent avant la conception des composants pour organiser ou figer une ou plusieurs utilisations.

Cependant, d'un point de vu du processus de vérification, la démarche que nous préconisons dans Kmelia est plutôt ascendante. De ce fait, nous avons consacré un effort considérable à la spécification et à la vérification des composites que nous détaillons dans la section suivante.

5.2.2 Phase de spécification

La phase de spécification comprend trois activités : conception de systèmes (assemblage/composition), spécification de composants et spécification de services. Dans une approche ascendante, l'activité de conception de système démarre en premier. Le spécifieur définit le système comme une collection de sous-systèmes et de composants interactifs. Si les composants ou les assemblages qui correspondent aux exigences du spécifieur existent déjà sur l'étagère, ils peuvent être directement intégrés dans la conception du système. Sinon, l'activité de spécification de composants produira de nouveaux composants. Une fois la structure de composant établie, l'activité de spécification de services débute. Le point principal de cette phase est que les contrats doivent être explicitement exprimés à chaque niveau afin d'être vérifiés par la suite.

5.2.2.1 L'apport des composites Kmelia

Que l'on adopte une approche ascendante ou descendante, le concepteur d'un composite a toujours besoin de sélectionner des composants parmi ceux qui sont disponibles pour construire son composite. Les services contenus dans ces composants n'ont pas été forcément conçus pour le besoin du concepteur du composite ; même si ceux-ci répondent à ses fins, ils peuvent être des opérations très génériques avec des pré/post-conditions très faibles ou des services spécifiques avec des pré/post-conditions fortes et plus précises.

Plusieurs propriétés peuvent être considérées lors de la conception d'un composite selon l'objectif (ou les objectifs) de son concepteur, à savoir :

- **la sûreté** qui nécessite la vérification des sous-composants individuellement et puis leur composition,
- **le passage à l'échelle** qui requiert une abstraction de plus haut niveau permettant ainsi de cacher le détail sur la structure des sous-composants et leurs services,
- **la flexibilité** qui exige la prise en compte des changements futurs dans le composite (par exemple, la substitution d'un sous-composant par un autre mieux adapté),
- **l'adaptation :** les services (et en particulier les services promus) doivent être adaptés à l'environnement cible du composite (qui n'est pas forcément celui pour lequel ils ont été conçus auparavant si le service est promu).

Pour atteindre ces objectifs, il pourrait être nécessaire de modifier les contrats des services d'origine (et donc changer les prédicats correspondants).

Dans la plupart des modèles, la promotion de service laisse les contrats de service (pré et post-conditions) inchangés ($Predicate_{origin} \Leftrightarrow Predicate_{promoted}$). Ce type de promotion est appelé *délégation pure* dans certains modèles [Oza07a]. Dans ce cas, la promotion est sûre car si le contrat est garanti au niveau du service origin alors il le sera forcément dans le service promoted au niveau du composite.

La promotion de services peut être classifiée selon plusieurs critères, à savoir :

- la nature du service : promotion d'un service offert ou requis,
- le changement du contrat : renforcement, affaiblissement ou aucun changement de prédicats.
- la sûreté : promotion sûre par construction, sûre par preuve, souvent non sûre (sauf sous quelques conditions) ou bien non sûre.

Dans la suite, nous classifions la promotion selon la nature du service promu et étudions les différentes combinaisons dues à la modification des contrats. Nous discutons également la sûreté de chaque cas.

5.2.2.2 Promotion des services offerts

Le tableau 5.2.1 résume les différents cas de changement de prédicats lors de la promotion d'un service offert, il montre également la sûreté de chaque cas.

- ***Affaiblissement de la pré-condition*** ($Pre_{origin} \Rightarrow Pre_{promoted}$) est, dans le cas général, non sûr car le contrat du service promu pourrait ne plus être valide. Il permet, par exemple, à l'appelant potentiel d'appeler le service promoted dans un contexte où le service origin ne peut pas garantir sa post-condition.
- ***Renforcement de la pré-condition*** ($Pre_{promoted} \Rightarrow Pre_{origin}$) garantit que la pré-condition du service original est toujours vraie. Ce cas est utile quand le composite CC est conçu pour être exécuté dans un environnement plus spécifique que celui de ses sous-composants génériques (*e.g.* sous-typage, restriction du domaine, *etc.*). Dans de tels cas, le renforcement de la pré-condition permet aussi d'adapter l'interface du composite et faciliter la substitution avec des services plus spécifiques.

- ***Affaiblissement de la post-condition*** ($Post_{origin} \Rightarrow Post_{promoted}$) est sûr car le service original garantirait plus que ce dont on a besoin au niveau du service promu. Ce type de promotion est utile quand le composite n'as pas besoin d'autant de précisions sur le résultat du service d'origine. Le fait d'avoir une post-condition plus faible permet de faciliter les substitutions possibles des services internes d'un composite. Du point de vue méthodologique, cela implique l'encapsulation de certaines informations jugées pas importantes au niveau du composite afin de faciliter les futures substitutions internes.
- ***Renforcement de la post-condition*** ($Post_{promoted} \Rightarrow Post_{origin}$) n'est pas sûr dans le cas général. Cependant, la partie du contrat modifiée dans ce cas est de la responsabilité du spécifieur du service promoted. Il y a des cas où il peut ajouter des contraintes sur le contexte d'exécution du service origin afin d'assurer sa post-condition $Post_{origin}$.

Dans la suite de cette section, nous allons caractériser tous les cas sus-mentionnés.

Post \ Pré	Affaiblie	Inchangée	Renforcée
Affaiblie	✗	✓	✓
Inchangée	✗	✓	✓
Renforcée	✗	!	!

✓ : promotion sûre,
✗ : promotion non sûre,
! : promotion souvent non sûre.

Tableau 5.2.1 – Modification des prédicats lors d'une promotion d'un service offert

Les cas non sûrs : Comme le montre le tableau 5.2.1, les trois cas où la pré-condition est affaiblie ne sont jamais sûrs. Cela nous permet d'avertir systématiquement le spécifieur que la promotion qu'il est en train d'exprimer ne peut être prouvée comme étant correcte.

Les cas sûrs : Les cas de promotion dans cette catégorie sont toujours sûrs du fait que le contrat est garanti au niveau du composite (cf. section 6.5.2.2).

Les cas souvent non sûrs La promotion dans cette catégorie n'est pas souvent sûre. Cependant il existe certains cas qui peuvent être non seulement sûrs mais utiles également.

1. ***Renforcer à la fois la pré et la post-condition*** *: "renforcement par paramètres"*. Cette promotion peut être sûre si on arrive à prouver que la restriction faite sur la pré-condition implique la nouvelle post-condition promue. Ce cas est illustré dans la section 6.5.2.2. Par ailleurs, bien que peu probable, la post-condition pourrait également être renforcée par le contexte (voir ci-dessous).
2. ***Maintenir la pré-condition et renforcer la post-condition*** *: "renforcement par contexte"*. Ce cas est sûr seulement lorsque l'exécution des sous-composants renforce l'invariant du composite en imposant des restriction sur

son état de telle sorte qu'il renforcera également la post-condition qui, elle aussi, dépend de cet état. Ce type de promotion ne peut être appliqué que si le modèle de composants supporte la notion d'observabilité (cf. section 4.4.1).

5.2.2.3 Promotion des services requis

Nous rappelons que contrairement au contrat d'un service offert, dans le contrat d'un service requis, la pré-condition spécifie ce que les appelants potentiels doivent garantir avant l'appel du service qui fournit le service requis. Inversement, la post-condition, quant à elle, spécifie ce qu'attendent les appelants du service requis (cf. figure 5.3).

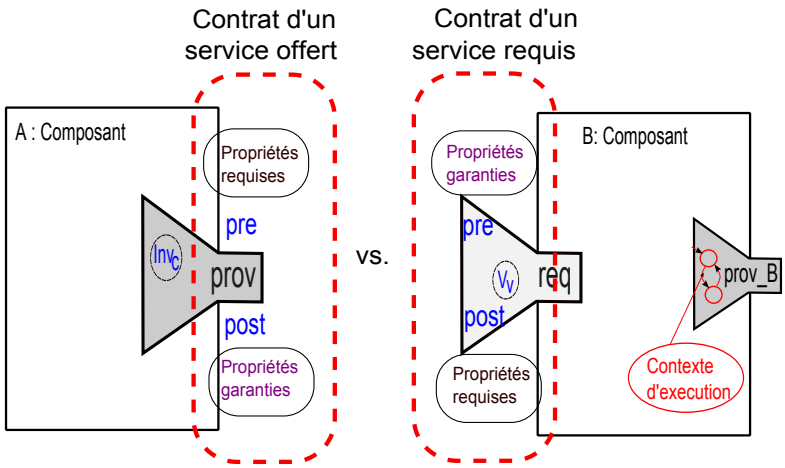


FIGURE 5.3 – Contrat d'un service offert *vs.* requis

Dans ce contrat, le client est l'appelant du service requis, et le fournisseur est le service qui offre ce dernier. C'est le cas contraire de ce que nous avons vu dans le tableau 5.2.1, avec une restriction supplémentaire qui réside dans le fait que le contexte d'exécution dans ce cas est dans le comportement des appelants et donc, contrairement au cas précédent, on ne peut pas le contraindre du fait qu'il n'est pas forcément connu, et par conséquent, il est plus difficile d'éviter les cas souvent non sûrs.

1. ***Maintenir la pré-condition et renforcer la post-condition.*** Cette stratégie est utilisée lorsque le service promu promoted exige plus de contraintes que celle du service origin. Cela permet, par exemple, d'uniformiser l'interface du composite.
2. ***Affaiblir la pré-condition et maintenir la post-condition.*** Dans ce cas, le service promu est moins sûr que le service d'origine. En revanche, cela permet d'anticiper le changement dans le service origin ou de cacher certains détails au niveau du composite.
3. ***Affaiblir la pré-condition et renforcer la post-condition.*** Cette stratégie combine les objectifs et l'utilité de (2) et (3).

Le Tableau 5.2.2 résume les différents cas de changement des assertions lors de la promotion d'un service requis et leur statut vis-à-vis de la sûreté.

Post \ Pré	Affaiblie	Inchangée	Renforcée
Affaiblie	✗	✗	✗
Inchangée	✓	✓	✗
Renforcée	✓	✓	✗

✓ : promotion sûre,
✗ : promotion non sûre,

Tableau 5.2.2 – Modification des assertions lors de la promotion d'un service requis

5.2.3 Phase de vérification

Les modèles produits au cours de la phase de spécification sont analysés en vérifiant les propriétés exprimées par les contrats. La phase de vérification du système est fondamentale puisqu'elle garantit que l'architecture définie statiquement lors de la phase de conception est en accord avec les besoins de l'utilisateur. Le processus de vérification (cf. figure 5.2) considère cinq activités d'analyse formelle, chaque activité référant aux contrats de la section 5.1.

1. L'analyse statique vérifie la correction syntaxique à tous les niveaux ainsi que l'accessibilité des services du niveau de composant, et l'interopérabilité statique du niveau d'assemblage qui, elle-même, couvre la compatibilité des signatures de services et la cohérence de la structure de services.
2. La correction fonctionnelle vérifie la propriété de cohérence comportementale au niveau du service et la conformité du comportement par rapport au contrat du service.
3. La vérification de cohérence couvre la propriété de cohérence du composant.
4. La compatibilité comportementale au niveau de l'assemblage.
5. La vérification de contrats d'assemblage et/ou de promotion garantit la conformité des services des composants assemblés ou encapsulés dans un composite.

Le tableau 5.2.3 met en évidence les différentes activités de vérification.

Contrat multi-niveaux	Niveau architectural			
Niveau de contrats	Service	Composant	Assemblage	Composite
Structurel	contrôle de type, dépendances de services	interface, contrôle de type	compatibilité de signatures, disponibilité de services	promotion, observabilité
Fonctionnel	correction fonctionnelle	préservation de l'invariant	compatibilité des pré/post	conformité des pré/post
Comportemental	absence d'interblocage, terminaison	cohérence du protocole	compatibilité comportementale	conformité comportementale
Propriétés	Correction	Cohérence	Clientèle	Encapsulation

Tableau 5.2.3 – Synthèse des propriétés exprimées par les contrats multi-niveaux

Dans les sections suivantes, nous nous intéressons à la mise en œuvre des différentes activités de vérification ci-dessus. Comme nous l'avons déjà indiqué dans le chapitre précédent, la première activité concernant la correction syntaxique est un prérequis pour le reste du processus. De plus, la quatrième activité (la vérification de la compatibilité comportementale) a été présentée dans la section 3.6.2. Ces deux activités sont déjà opérationnelles dans la plate-forme COSTO et ne seront donc pas détaillées ici.

En effet, nous avons identifié deux niveaux de prise en compte des contrats. Un niveau local (composants individuels avec leurs services) et un niveau global du système (assemblages et compositions). Ces vérifications s'intègrent sous forme d'obligations de preuve [Dij78].

Définition 5.2 *Une obligation de preuve est une formule mathématique à démontrer afin d'assurer qu'un élément de modélisation (service, composant, assemblage ou composite) est correct.*

Dans cette optique, les obligations de preuve sont une aide au processus de vérification proposé plus haut [MAA10a]. Les vérifications intra-composant font office de vérification *unitaire* des composants. Cette étape de vérification sera mise en place très tôt dans le processus de développement (*design time*). Les vérifications inter-composants permettent une vérification de des composants mis ensemble (assemblages et composites). Cette étape peut débuter une fois que les composants simples ont été vérifiés. Dans la suite de ce chapitre, nous détaillons les deux niveaux d'obligations de preuve.

5.3 Obligations de preuve intra-composant

Nous proposons principalement trois obligations de preuve locales à un composant. La première, au niveau des services du composant, permet de vérifier que le comportement de chaque service est conforme à son contrat (pré/post-condition). La seconde, au niveau du composant, permet de vérifier que l'invariant du composant est préservé par son implantation (l'ensemble des services qu'il offre). La troisième, également au niveau du composant, permet de s'assurer de la cohérence d'un protocole du composant.

5.3.1 Contrat de service

Rappelons que dans Kmelia, un service *serv* est défini, entre autres, par un comportement dynamique $\mathcal{B}$ (cf. section 3.4.2). La première vérification concerne la validité du comportement vis-à-vis du contrat du service $Cont = \langle Pre, Post \rangle$. Pour cela, nous montrons que lorsqu'on fournit au composant des valeurs d'entrées autorisées par la pré-condition du service, alors le composant produit bien des sorties qui sont conformes à la post-condition. Ce contrat inclut :

- La *cohérence comportementale* qui assure que l'exécution des actions de service ne conduit pas à des états incohérents (cf. section 3.6.2) et que le service se terminera toujours.

– La *correction fonctionnelle* qui exprime le fait qu'un service réalise ce qu'il est censé faire. La correction fonctionnelle d'un service est définie ici en utilisant la spécification à la Hoare (pré-condition, actions, post-condition) où les actions représentent le comportement du service. Cette propriété doit être vérifiée en tenant compte des propriétés du composant (notamment celles exprimées dans l'invariant).

Autrement dit, pour prouver la correction d'un service, il faut montrer que le service termine pour toutes ses entrées valides (*terminaison*) et que lorsqu'il s'arrête il donne un résultat correct (*correction fonctionnelle*).

$$\boxed{correction\ de\ service = terminaison \wedge correction\ fontionnelle}$$

Dans la suite, nous nous intéresserons uniquement à la preuve de la correction fonctionnelle basée sur les travaux de Hoare et Dijkstra [Hoa69, Dij75]. Ainsi nous supposons la terminaison des services considérés.

Obligation de preuve

Définition 5.3 (Correction fonctionnelle CF) *Un service* $serv = \langle Pre, \mathcal{B}, Post \rangle$ *est correct par rapport à son contrat* Pre *et* $Post$ *si pour tout état initial vérifiant* Pre*, si l'exécution de* $\mathcal{B}$ *se termine, alors* $Post$ *est vraie après l'exécution de* $\mathcal{B}$*.*

La définition ci-dessus peut être exprimée par *le triplet de Hoare* : $\{Pre\}\mathcal{B}\{Post\}$ qui signifie que $Post$(la post-condition) est vraie dans chaque état atteint par l'exécution de $\mathcal{B}$ à partir d'un état initial dans lequel Pre (la pré-condition) est vraie. La vérification de la *correction fonctionnelle* est basée sur l'algorithme du calcul de la plus faible pré-condition, noté $\mathcal{WP}$ (*weakest pre-condition*) [Dij75]. Le calcul de la plus faible pré-condition à partir de $Post$ et $\{\mathcal{B}\}$ permet de prouver le triplet Pre $\{\mathcal{B}\}$ $Post$. En effet, Si $P \Rightarrow Q$ alors Q est dit plus faible [1] que P.

L'algorithme de calcul de la plus faible pré-condition pour le comportement $\mathcal{B}$ d'un service transforme le prédicat de la post-condition $Post$ en un prédicat de pré-condition $\mathcal{WP}$ $\{\mathcal{B}\}$ $Post$. Ce dernier est le prédicat le *plus faible* assurant que si un état le satisfait alors, après exécution de $\mathcal{B}$, le prédicat $Post$ est vrai. Le triplet de Hoare est donc vérifié si et seulement si $Pre \Rightarrow \mathcal{WP}$ $\{\mathcal{B}\}$ $Post$. Par exemple, calculer $\mathcal{WP}$ $\{a1; a2\}$ $Post$ procède en arrière à partir de la post-condition $Post$, en calculant d'abord $\mathcal{WP}$ $\{a2\}$ $Post$, en appliquant ensuite $\mathcal{WP}$ $\{a1\}$ au prédicat résultant.

Afin de simplifier la présentation de l'algorithme, on associe à chacun des états s_i du comportement $\mathcal{B}$, une propriété φ_i dite *invariante* au sens où la propriété φ_i est vraie à chaque fois que le déroulement du service $serv$ passe par l'état s_i. On associe particulièrement la post-condition $Post$ à tous les états finaux s_f du $\mathcal{B}$ c'est-à-dire $\varphi_f \equiv Post$. La propriété φ_f garantit ainsi que lorsque l'exécution du service atteindra un état final, alors la propriété $Post$ (qui nous intéresse ici) sera valide.

La sémantique de base de l'algorithme du calcul de la plus faible pré-condition est basée sur la règle suivante : $\mathcal{WP}\{V := E\}Q = Q[E/V]$. Où V est une variable,

1. ou P est dit plus fort que Q et donc on parle de générateur de plus forte post-condition.

E une expression et Q un prédicat, il suffit donc de remplacer V par E dans Q. La correction fonctionnelle d'un service peut donc être prouvée inductivement en partant de la post-condition et en calculant la plus faible pré-condition selon les règles présentées dans le tableau 5.3.1.

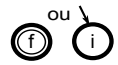
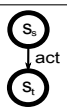
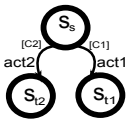
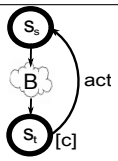
Structure	Calcul du plus faible prédicat
	$\mathcal{WP}(f) = \varphi_f$ (ou $\mathcal{WP}(i) = \varphi_i$)
	$\mathcal{WP}(s_s) = \varphi_t[act]$
	$\mathcal{WP}(s_s) = (\varphi_{t1}[act1] \ \wedge \ C1) \vee (\varphi_{t2}[act2] \ \wedge \ C2))$
	$\mathcal{WP}(s_s) = (\varphi_{t1}[B]) \vee (\varphi_{t2}[act2] \wedge \ \neg C)$

Tableau 5.3.4 – Règles définissant le calcul de la plus faible pré-condition ($\mathcal{WP}$).

Par exemple, la plus faible pré-condition du service de l'exemple de la figure 5.4 est calculée en procédant vers l'arrière depuis la dernière transition du comportement et en remplaçant dans la post-condition courante les variables par les valeurs qui leur sont affectées. Pour prouver le triplet de Hoare $\{x = 0\}x := x + 1 \ ; x := x + 2 \ ; \{x = 3\}$, il faut montrer que $x = 0 \Rightarrow \mathcal{WP} \ \{\mathcal{B}\}(x = 3)$ c'est-à-dire que $x = 0 \Rightarrow ((x + 1) + 2) = 3$ ce qui est trivialement vérifié.

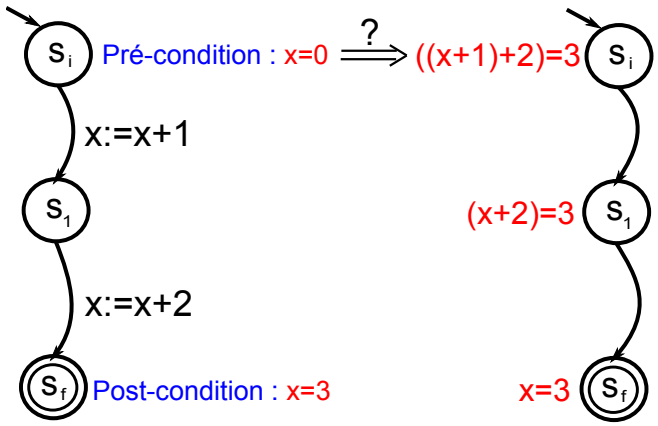


FIGURE 5.4 – Exemple de calcul de la plus faible pré-condition.

Notons que les règles définies ci-dessus ne s'appliquent que sur un sous-ensemble des eLTS considérés dans cette thèse, étant donnée la grande difficulté à traiter tout

type de eLTS. Cette difficulté est essentiellement due au fait de la présence de boucles non-structurées et de types de données avancés. En effet, une boucle dans un eLTS ne peut pas forcément être transformée en une structure de contrôle comparable aux boucles que nous trouvons dans la plupart des langages de programmation. Même pour ces dernières il n'y a, ni en théorie, ni en pratique, d'algorithme générale pour calculer leur plus faible pré-condition. En effet, la réécriture avec les règles ci-dessus peut ne pas se terminer. Une approche a été proposée pour pallier ce problème. Elle consiste à utiliser des invariants de boucle qui peuvent être fournis par le spécifieur ou par un algorithme de génération d'invariants.

5.3.2 Contrat de composant

Le contrat de composant est un accord entre le composant et ses clients. Il prévoit qu'un composant peut être (ré)utilisé en toute confiance. Il traite de deux propriétés : la préservation de l'invariant du composant (garantie) et la cohérence du protocole [1] de ce dernier (hypothèse).

- La propriété de préservation de l'invariant du composant permet de garantir que les propriétés invariantes de ce dernier sont conservées par l'ensemble des services incorporés dans le composant [2]. Considérant qu'un composant équipé de services est cohérent si ses propriétés sont satisfaites quel que soit le comportement des services, on peut définir un contrat de préservation de cohérence entre les services et leur composant pour s'assurer de cette propriété.
- La propriété de cohérence du protocole exprime le fait que l'ordre dans lequel les services doivent être invoqués par les clients est correct en terme de règles données par la spécification de services.

Dans la suite de cette section, nous détaillons les obligations de preuve liées à ces deux propriétés.

Obligations de preuve

Définition 5.4 (Préservation de l'invariant) *Il s'agit de garantir que les services fournis et requis ne violent pas l'invariant de leur composant. Nous considérons cette préservation à trois niveaux d'observabilité selon les obligations de preuve suivantes* ***CO, CC*** *et* ***CV****.*

$$(i)\ \boxed{Inv^{O}(before) \wedge Pre^{O}(before) \wedge Post^{O}(before, after) \Rightarrow Inv^{O}(before, after)}\ (\mathbf{CO})$$

Où : $before$ (resp. $after$) fait référence aux valeurs de l'espace d'état du composant juste avant l'appel du service (resp. juste après la terminaison du service) et l'indice ^{O} fait référence aux éléments observables (cf. section 4.4.1). Cette obligation de preuve assure que la partie observable du composant doit être cohérente,

1. Un protocole de composant est défini ici comme l'ensemble de toutes les séquences valides d'appels de service.

2. Un travail basé sur le raffinement des systèmes de transition a été proposé dans [KL04].

c'est-à-dire que les parties observables des pré/post-conditions d'un service offert sont suffisantes pour établir la partie observable de l'invariant.

$$(ii)\ \boxed{\begin{array}{c} Inv^O(before) \wedge Inv(before) \wedge Pre^O(before) \wedge \\ Post^O(before, after) \wedge Post^l \Rightarrow Inv^O(before, after) \wedge Inv(before, after) \end{array}}\ \textbf{(CC)}$$

De la même manière, cette obligation de preuve permet d'assurer que l'invariant complet (partie observable et non-observable) est préservé par les services offerts. $Post^l$ est la post-condition locale (non-observable) qui, elle aussi, doit être prise en compte pour établir la préservation de l'invariant. Notons qu'une variante de cette obligation est également définie pour les services internes. Ces derniers n'ont, par définition, pas de partie observable, donc il suffit juste de remplacer leurs pré/post-conditions observables dans l'obligation par les pré/post-conditions internes de ces services.

$$(iii)\ \boxed{Inv^V(before) \wedge Pre^V(before) \wedge Post^V(before, after) \Rightarrow Inv^V}\ \textbf{(CV)}$$

Où l'indice ^{V} fait référence aux éléments virtuels d'un service requis. Cette obligation de preuve concerne la cohérence du contexte virtuel 4.4.2, c'est-à-dire que chaque service requis doit préserver son invariant (virtuel) correspondant.

Ces obligations doivent être prouvées pour assurer la cohérence de l'invariant des composants Kmelia.

Cohérence du protocole. Un protocole Kmelia fait office de mode d'emploi pour les composants [AAA07b]. Il décrit, à travers une composition de service, les enchaînements licites de services. Par exemple, pour utiliser correctement les services read et write offerts par le composant FileServer, il faut d'abord invoquer le service open et ne pas oublier d'invoquer le service close à la fin. Un protocole pour un tel composant permet de spécifier que l'ordre d'invocation de ces différents services garantit une utilisation valide. Il peut s'exprimer ainsi : connexion.(read | write)*. disconnection en utilisant le formalisme des expressions rationnelles.

Dans la mesure où un protocole est considéré dans Kmelia comme un service offert, nous pouvons le soumettre aux mêmes vérifications que n'importe quel autre service offert. La détection d'incohérence dans les protocoles fait partie des vérifications nécessaires pour assurer la correction d'un composant. Un protocole d'un composant est *incohérent* si un des enchaînements de services qu'il décrit peut être impossible. Cela peut être dû :

- soit à l'existence de chemins gardés sans alternatives menant à l'état final du protocole, et que l'expression de ces gardes ne peut être évaluée à vrai ;
- soit à l'existence d'une séquence d'appels $s_1; ...; s_n; s_{n+1}$ telle que la prise en compte des post-conditions de s_1 à s_n implique la négation de la pré-condition de s_{n+1}, c'est-à-dire qu'un des services appelés avant s_{n+1} et dont les effets n'ont pas été remis en cause empêche le déroulement correct de s_{n+1} : il s'agit d'une séquence infaisable. Par exemple, si le service `connexion` possède une pré-condition `not(connected)` et une post-condition `connected`, la séquence

`connexion; connexion` rend le protocole incohérent, ainsi que toute autre séquence ne contenant aucune modification de `connected` entre deux appels de `connexion`.

Le principe de la vérification du protocole est donné dans la définition suivante.

Définition 5.5 (Cohérence du protocole) *On considère un protocole et les séquences d'appels de services allant d'un état initial à un état final du protocole pour éviter les boucles. Pour chacune des séquences, on vérifie que, pour toutes les sous-séquences $s_i; s_j$ (avec $j = i + 1$), la condition dans laquelle s_i est achevé n'empêche pas le déroulement de s_j.*

$$\boxed{\neg(Post_{s_i} \Rightarrow \neg Pre_{s_j})}$$

Plusieurs cas de figure peuvent éventuellement poser problème :

- Si un des protocoles est interruptible, tout service présent dans l'interface peut être appelé à chaque état du protocole. Cela peut provoquer une explosion combinatoire dans la recherche de séquences incohérentes dans le protocole.
- Lorsqu'un service s_i est le seul qui permette d'établir la pré-condition d'un autre service s_j alors s_i doit apparaître dans le protocole d'utilisation (et avant s_j).

Du point de vue méthodologique, afin d'éviter des incohérences dues aux interférences avec l'état d'un composant, il est déconseillé d'avoir dans l'interface du composant les services opérant des modifications d'une partie de l'état prise en compte dans les pré-conditions de services contrôlés par des protocoles interruptibles.

5.4 Obligations de preuve inter-composants

La vérification inter-composants concerne les propriétés que l'on souhaite vérifier lorsque des composants sont connectés les uns aux autres. On propose principalement deux vérifications inter-composants : la première permet de vérifier que deux composants sont compatibles, c'est-à-dire qu'ils peuvent être assemblés selon un contrat préétabli (cf. section 5.4.1) ; la seconde permet de vérifier la conformité des services promus au niveau des composites par rapport aux services d'origine, c'est-à-dire qu'ils préservent toujours leur contrat fonctionnel (cf. section 6.5.2.2).

5.4.1 Contrat d'assemblage

En supposant que les composants individuels aient les propriétés souhaitées, on souhaite déterminer statiquement les propriétés des assemblages. On traite ici de la cohérence et de la composabilité. La cohérence d'un assemblage est primordiale pour son bon fonctionnement. En dehors de la cohérence au niveau des interfaces des services qui sont reliées pour former l'assemblage, une bonne conception de l'assemblage doit permettre de détecter des incohérences au-delà des simples correspondances de paramètres et de types. Dans la proposition Kmelia, les assertions (pré/post-conditions) sont utilisées pour mettre en place des contrats entre les services dans les assemblages.

Nous analysons la bonne description des assemblages en nous basant sur la notion de composabilité. De façon synthétique, quatre niveaux de compatibilité sont pris en compte :

1. Signature : le profil des services doit être compatible.
2. Dépendances : les interfaces de composants et les dépendances de services doivent être compatibles.
3. Contrats (pré/post-conditions) : le service offert "remplit le contrat" défini par le service requis.
4. Interactions : les communications entre services doivent être cohérentes et ne pas aboutir à des blocages.

Du fait de la description des services, la compatibilité lors des assemblages de composants est vérifiée de façon progressive. Une première étape de la vérification de la compatibilité statique est faite à la compilation de la spécification Kmelia par analyse des interfaces en profondeur : vérification des signatures de tous les services en jeu, concordance des liens et sous-liens, complétude des services requis pour tout service offert dans l'assemblage (seuls sont traités les services effectivement utilisés dans l'assemblage). Lorsque cette première étape de compatibilité est terminée, les assertions des services liés sont utilisées pour s'assurer du respect du contrat d'assemblage. Le quatrième niveau consiste à vérifier la compatibilité comportementale entre les différents services impliqués dans l'assemblage (cf. section 3.6.2).

Dans la suite de cette section, nous traitons les obligations de preuve liées au troisième niveau (contrats pré/post-conditions).

Obligation de preuve

Définition 5.6 (Contrat de clientèle) *Soit un service requis servR (d'un composant C) avec les assertions Pre^V et $Post^V$; soit un service offert servP avec les assertions Pre et Post. Le service servP permet de satisfaire le requis servR lorsque :*

$$\boxed{\begin{array}{lcl} Pre^V_{servR} & \Rightarrow & Pre^O_{servP} \\ & \wedge & \\ Post^O_{servP} & \Rightarrow & Post^V_{servR} \end{array}} \qquad \textbf{(CA)}$$

De cette façon, on décrit un contrat de telle sorte que :
- les composants qui peuvent être assemblés avec C sont ceux qui sont à même de respecter le contrat du service requis servR ;
- les services du composant C qui requièrent servR connaissent les conditions dans lesquelles ils obtiennent de bons résultats.

Proposition 5.1 *En prenant en compte l'espace d'état virtuel du service requis et l'espace d'état observable du service servP, le précédent contrat d'assemblage s'étend comme suit :*

1. $\boxed{Inv^V \wedge Inv^O \wedge Map \wedge Pre^V_{cm} \Rightarrow Pre^O}$ **(CA)**
2. $\boxed{Inv^O(before) \wedge Post^O(before, after) \Rightarrow (Pre^V_{cm}(before) \Rightarrow Post^V_{cm}(before, after))}$

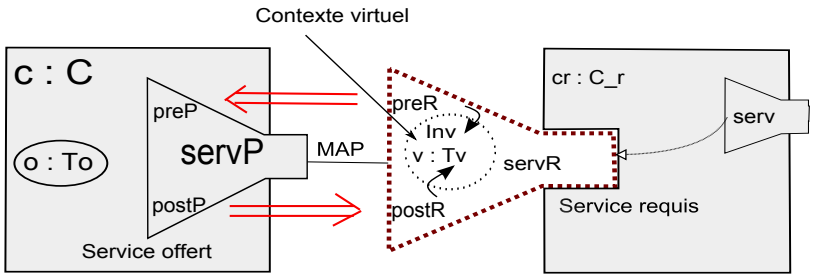


FIGURE 5.5 – Contrat de clientèle.

L'indice $_{cm}$ (pour *context mapping*) indique qu'il peut être nécessaire d'appliquer une correspondance entre les variables du contexte virtuel et celles de l'espace d'état du composant offrant le service (cf. section 4.4.2). Le prédicat $(Pre^V_{cm}(before) \Rightarrow Post^V_{cm}(before, after))$ modélise le contrat fonctionnel du service requis, car dans Kmelia, un service requis exprime non seulement les exigences qu'il attend $(Post^V)$ mais également les garanties qu'il assure à son fournisseur (Pre^V). Cette exigence d'analyse de cohérence au niveau des services est aussi traitée au niveau de la promotion des services requis et offerts que nous allons détailler dans section suivante.

5.4.2 Contrat de composite

Que l'on adopte une approche ascendante ou descendante, le concepteur d'un composite a toujours à sélectionner des composants parmi ceux qui sont disponibles pour construire son composite. Les services contenus dans ces composants n'ont pas été forcément conçus pour le besoin du concepteur du composite ; même si ceux-ci répondent à ses fins, ils peuvent être des opérations très génériques avec des pré/post-conditions très faibles ou des services spécifiques avec des pré/post-conditions fortes et plus précises. Il pourrait être nécessaire de modifier les contrats des services d'origine (et donc changer les prédicats correspondant à leurs contrats). Dans ce cas, nous avons proposé une obligation de preuve pour assurer la sûreté de la promotion d'un service lorsque le concepteur d'un composite décide de modifier le contrat du service d'origine.

Comme nous nous plaçons dans un cadre de vérification compositionnelle, à ce niveau, les obligations de preuve liées au services, composant et assemblage inclus dans le composite doivent déjà être réalisées. L'obligation associée à la sûreté de la promotion sera détaillée dans la suite de cette section.

Obligation de preuve

Considérons un composant (composite) CC contenant deux sous-composants CA et CB. Nous étudions la promotion exprimée par le lien de promotion entre le service offert origin du composant CA et le service promoted du composite CC (cf. figure 6.5).

Nous rappelons que le contrat du service origin garantit que si la pré-condition Pre_{origin} du service origin et l'invariant du composant CA sont vrais, alors le service

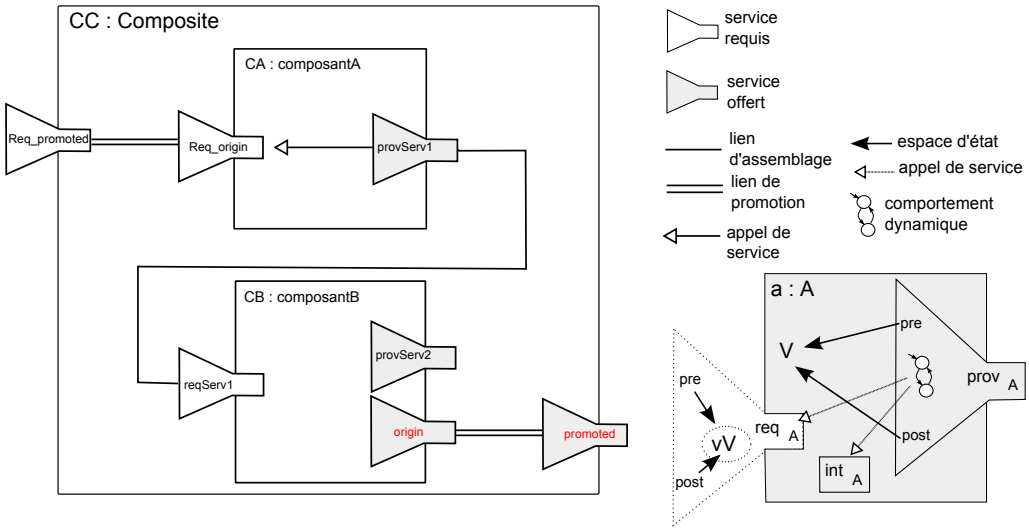


FIGURE 5.6 – Promotion au niveau d'un composite

origin satisferait sa post-condition $Post_{origin}$. Ce contrat peut être exprimé comme suit :

$$\boxed{Pre_{origin}(before) \wedge inv_C(before) \Rightarrow Post_{origin}(after) \wedge inv_C(after)}$$

Hypothèse 5.2 *La pré-condition Pre_{origin} doit être établie par l'appelant du service (le client), tandis que la post-condition $Post_{origin}$ et l'invariant Inv_{CA} doivent être établis par l'appelé (le fournisseur).*

Hypothèse 5.3 *Le contrat ci-dessus est garanti pour le service origin et le comportement de ce dernier est conforme à son contrat.* [1]

Définition 5.7 (Contrat de promotion) *Sous les hypothèses 5.2 et 5.3, la promotion du service origin est considérée correcte si et seulement si le contrat de clientèle du service promu promoted est correct.*

$$\boxed{Pre_{promoted}(before) \wedge inv_{CC}(before) \Rightarrow Post_{promoted}(after) \wedge inv_{CC}(after)} \quad \textbf{(CP)}$$

La promotion est dite *correcte* si on est capable de prouver le contrat de clientèle pour le service promoted. Pour certains types de promotion, nous pouvons exploiter une bonne partie de la preuve déjà faite au niveau du service origin et simplifier largement le problème de correction des promotions. Néanmoins, le fait d'identifier le type de la promotion en question est, dans le cas général, plus difficile que le problème de correction lui-même. C'est pourquoi nous proposons, dans la suite, des opérateurs pour expliciter l'intention du spécifieur.

1. Autrement dit, nous ignorons le fait que le service soit sous-spécifié et nous ne cherchons pas non plus à rendre ses pré/post-conditions plus précises.

5.5 Conclusion

Dans ce chapitre, nous avons montré les apports des contrats introduits dans le chapitre 4. D'abord, nous avons introduit la notion de contrats multi-niveaux pour maîtriser la construction progressive des systèmes à base de composants et faciliter le processus de leur vérification. En effet, la déclinaison en couche des contrats permet la répercussion des résultats vers les couches de plus bas niveau pour éviter des vérifications inutiles. Nous avons également montré les apport des composites Kmelia en terme de flexibilité dans leur spécification et de facilité dans leur vérifications. Ensuite, nous avons détaillé les obligations de preuve selon deux niveaux de prise en compte des contrats : le premier au niveau local des composants et de leurs services (intra-composant), et le second au niveau global des assemblages et des compositions (inter-composants).

Ces obligations de preuve sont une aide aux activités de vérification dans la processus de développement que nous avons préconisé dans ce chapitre. Les obligations de preuve intra-composant facilitent la vérification *unitaire* des composants. Cette étape de vérification sera mise en place très tôt dans le processus de développement (*design time*). Les obligations de preuve inter-composants permettent une vérification compositionnelle des *assemblages* et des composites c'est-à-dire en se basant sur les résultats des analyses effectués au niveau intra-composant. Cette étape a donc intérêt de ne débuter qu'une fois que les composants simples ainsi que leurs services ont été vérifiés. Le chapitre suivant présente la façon dont ce processus est mis en pratique dans le contexte de Kmelia/COSTO.

Chapitre 6

Mise en œuvre des vérifications à l'aide de la méthode B

Dans ce chapitre, nous nous intéressons à l'automatisation du processus de la vérification des propriétés (des composants et de leur composition) exprimées par les contrats multi-niveaux introduits dans le chapitre précédent. Ce processus est mis en œuvre grâce à l'utilisation de la méthode B [Abr96]. Une introduction à la méthode B est ainsi donnée dans la section 6.1. La section 6.2 présente une vue globale de notre approche de vérification. La section 6.3 expose nos propositions de schémas de traduction de Kmelia vers B. Les sections 6.4 et 6.5 respectivement, montrent comment ces schémas de traduction sont exploités pour la vérification intra-composant et la vérification inter-composants (c'est-à-dire les contrats d'assemblage et de promotion).

6.1 Aperçu de la méthode B et ses outils

B est une méthode formelle pour analyser, spécifier, concevoir et implanter des systèmes logiciels [Abr96]. Elle permet de décrire des modèles abstraits de logiciels à l'aide de la théorie des ensembles et de la logique du premier ordre étendue avec la théorie des substitutions généralisées. À partir de ces modèles abstraits appelés machines abstraites, B propose des techniques de raffinements successifs aboutissant au code exécutable. En B, en vue de construire des modèles et des logiciels corrects, des preuves de cohérence des machines et de leurs raffinements sont effectuées. La méthode B génère des obligations de preuve pour chaque niveau de raisonnement. Des outils de preuve existent pour effectuer automatiquement ou de façon interactive ces preuves. La méthode B est toujours en évolution. Actuellement on peut l'utiliser sous sa version dite classique mais aussi sous la version Event-B (ou B événementiel) pour la spécification de systèmes distribués [ABHV06, AH07b, Abr10]. Le cadre de preuves de propriétés et de raffinements successifs n'a pas fondamentalement changé mais a été étendu.

6.1.1 Machines abstraites B

La structure d'une machine abstraite B est présentée dans le listing 6.1. Une machine abstraite B comporte, en dehors des paramètres et de diverses clauses de structuration, les descriptions d'une partie statique et d'une partie dynamique. La partie statique spécifie un espace d'états à l'aide d'une liste d'ensembles, de définitions, de constantes, de variables et d'un invariant (I) qui est un prédicat exprimant les propriétés globales de tous les états du système spécifié ; l'invariant capture ainsi la sémantique voulue par le spécifieur. Ces différentes descriptions apparaissent dans des clauses comme dans les exemples que nous introduirons.

Listing 6.1 – Structure d'une machine abstraite de la méthode B

```
MACHINE
    M(X,x) /* Déclaration du nom de la machine abstraite (M) et de la liste des
              paramètres formels éventuels ∈ ensembles (X) ou valeurs (x) */
CONSTRAINTS
    C      /* Déclaration des propriétés logiques des paramètres ensembles (X)
              ou valeurs (x) de la machine */

SETS
    s      /* Déclaration des définitions des ensembles abstraits ou énumérés de
              la machine */

CONSTANTS
    c      /* Déclaration des identificateurs des constantes */

PROPERTIES
    P      /* Déclaration des propriétés logiques des ensembles et constantes (c)
              déclarés, avec typage et expression d'évaluation des constantes */

VARIABLES
    v      /* Déclaration des identificateurs des variables */

INVARIANT
    I      /* Déclaration des propriétés logiques invariantes des variables (v)
              déclarées, avec typage des variables */

ASSERTIONS
    J      /* Déclaration des assertions logiques sur les variables de la machine */

INITIALISATION
    U      /* La substitution généralisée initialisant les variables de la machine */

OPERATIONS /* Liste des opérations */
    res ⟵ op (param) =   /* Signature de l'opération*/
    PRE Pre              /* Pré−condition de l'opération op */
    THEN S               /* Corps de l'opération op */
    END;                 /* Fin de l'opération op */
    ...

END        /* Fin de la définition de la machine */
```

La partie dynamique décrit les opérations fournies par la machine. Les opérations d'une machine B, dont nécessairement une initialisation (U), sont modélisées chacune par des substitutions généralisées S qui peuvent avoir une pré-condition (Pre). Les substitutions généralisées sont des transformateurs de prédicats, à la *weakest-precondition* de Dijkstra).

6.1.2 Raffinement

Le raffinement concrétise une machine plus abstraite. Il concerne les données et les opérations. Du point de vue syntaxique, le raffinement est structuré comme une machine abstraite, la différence apparaît dans la déclaration même du raffinement, dans laquelle **MACHINE** est remplacée par **REFINEMENT**. De plus, un raffinement doit posséder une clause **REFINES** qui référence la machine abstraite à raffiner (cf. listing 6.2), et les mêmes signatures d'opération (les mêmes noms d'opérations avec les mêmes noms de paramètres) que sa machine abstraite.

Listing 6.2 – En-tête d'un raffinement B

```
REFINEMENT
    N       /* Déclaration du nom du raffinement */
REFINES
    M       /* Nom de la machine (ou du raffinement) à raffiner */
...
END         /* Fin de la définition du raffinement*/
```

La sémantique d'un raffinement, quant à elle, est différente de celle d'une machine abstraite, chaque raffinement inclut un invariant de liaison (ou *invariant de collage*) entre les variables qu'il contient et celles de la machine abstraite qu'il raffine ; cet invariant permet d'établir et de prouver que le raffinement est correct par rapport à la machine raffinée.

6.1.3 Preuve de cohérence

En plus de l'analyse syntaxique et du contrôle du type, la cohérence d'une machine abstraite est établie par la vérification de certaines propriétés liées à la spécification de la machine elle-même. Ces propriétés sont dites obligations de preuve (*POs proof obligations*). Plusieurs types d'obligations de preuve sont imposés par la méthode B et les obligations de preuve elles-mêmes varient en fonction du type du module considéré (machine, raffinement, *etc.*). Dans la suite, nous nous intéressons uniquement à celles que nous réutiliserons dans le contexte de notre travail.

Définition 6.1 (*PO* des assertions) *Les assertions de la machine doivent être démontrées à partir de l'invariant et des propriétés.*

$$\boxed{P \wedge I \Rightarrow J}$$

L'emplacement des assertions dans le texte de la spécification B est important puisque les assertions sont démontrées dans l'ordre, en ajoutant en hypothèses les assertions précédentes déjà démontrées.

Définition 6.2 (*PO* de l'initialisation) *Cette obligation de preuve assure qu'un état initial obtenu après l'exécution de l'initialisation U préserve l'invariant I de la machine.*

$$\boxed{I \wedge P \wedge J \Rightarrow [U]I}$$

Définition 6.3 (*PO* d'une opération dans une machine) *Cette obligation de preuve permet de vérifier que si une opération est appelée sous la pré-condition Pre alors elle préserve l'invariant, c'est-à-dire que si l'invariant était vrai avant l'exécution de l'opération, alors il est toujours vrai après.*

$$\boxed{P \wedge J \wedge I \wedge Pre \Rightarrow [S]I}$$

Définition 6.4 (*PO* d'une opération dans un raffinement) *Une opération définie dans un raffinement est correcte lorsqu'elle préserve l'invariant sans contredire l'opération qu'elle raffine, et lorsque sa pré-condition est moins restrictive que la pré-condition spécifiée. Elle dépend non seulement des deux opérations (abstraite (resp. concrète) notée S_{abs} (resp. S_{ref})) mais également des deux invariants (l'invariant de la machine abstraite I_{abs} et celui du raffinement I_{ref}).*

$$\boxed{P \wedge J \wedge I_{abs} \wedge Pre_{abs} \wedge I_{ref} \Rightarrow Pre_{ref} \wedge [S_{ref}]\neg[S_{abs}]\neg I_{ref}}$$

Cette obligation de preuve permet de montrer qu'il existe une évaluation des variables de la machine abstraite compatible avec l'opération abstraite et qui vérifie l'invariant défini au niveau du raffinement. Autrement dit, à chaque exécution de l'opération concrète, il existe une exécution de l'opération abstraite qui va vers un état valide pour l'invariant concret.

Par exemple, si l'opération abstraite retourne une valeur dans l'intervalle [1..10], alors la nouvelle opération pourra retourner une valeur dans l'intervalle [2..4]. Comme pour les opérations de machines abstraites, le but à prouver se base sur l'invariant du composant ; dans le cas des raffinements, on y ajoute (par conjonction) un prédicat de liaison entre les variables de sorties de l'opération du raffinement, et les variables de sorties renommées de l'opération spécifiée.

Les obligations de preuve présentées ci-dessus sont systématiquement générées par les outils associés à B.

6.1.4 Outils

De nombreux outils ont été développés pour assister le développement basé sur la méthode B. Ces outils permettant, entre autres, de vérifier la cohérence du modèle abstrait et la conformité de chaque raffinement avec le modèle supérieur (prouvant ainsi la conformité de l'ensemble des implantations concrètes avec le modèle abstrait). Par exemple :

- AtelierB, outil industriel distribué gratuitement par la société Clearsy. Il fonctionne sous Linux, Windows, MacOs et Solaris.
- B4free [1] est le cœur de l'AtelierB, s'utilise avec Click'n Prove
- Bart [2], outil de raffinement automatique
- RODIN [3], la plate-forme B-évènementiel basée sur Eclipse
- B-Toolkit [4], outil développé par la société B-Core

1. http ://www.b4free.com/
2. http ://www.tools.clearsy.com/index.php5 ?title=BART_Project
3. http ://www.event-b.org/
4. http ://www.b-core.com/ONLINEDOC/BToolkit.html

Nous nous intéressons particulièrement à l'AtelierB [1], de la société ClearSy. Il permet de gérer un ensemble de composants (**MACHINE**, **REFINEMENT** ou **IMPLEMENTATION**) constituant un projet B. Les fonctionnalités principales concernent la vérification de la syntaxe et le contrôle de type, la génération des obligations de preuve, l'assistance à la preuve, et la génération du code exécutable.

L'outil de preuve permet de tenter une preuve automatique des obligations de preuve générées. Lorsque le prouveur automatique ne réussit pas à prouver une formule logique, le prouveur interactif permet à l'utilisateur d'orienter la preuve en fournissant des hypothèses ou en choisissant d'autres stratégies de preuve (par contradiction, par cas, *etc.*).

Un des avantages de la méthode B, est la souplesse qu'elle offre pour modéliser et prouver des propriétés exprimées sous forme de prédicats de la logique du premier ordre. Nous allons exploiter cette possibilité dans la technique de vérification des spécifications Kmelia que nous proposons dans la suite de ce chapitre.

6.2 Principes et objectifs de notre approche

Dans le chapitre précédent, nous avons établi un ensemble d'obligations de preuve pour vérifier diverses propriétés exprimées par des contrats fonctionnels (cf. tableaux 6.2.1).

Identifiant	Description de la propriété
(CC)	préservation de l'invariant par les services offerts.
(CO)	préservation de l'invariant observable par les services offerts.
(CV)	cohérence du contexte virtuel d'un service requis.
(CF)	correction fonctionnelle.
(CA)	cohérence des contrats d'assemblage (contrat de clientèle).
(CP)	cohérence des contrats de promotion.

Tableau 6.2.1 – Obligations de preuve liées aux contrats en Kmelia

La correction des composants concerne en partie la cohérence des services par rapport aux propriétés du composant qui les contient ; la correction des assemblages concerne, elle, la cohérence des liaisons effectuées entre les services requis et offerts de différents composants. La méthode que nous avons adoptée dans ce cas est basée sur le principe suivant : un service offert est un raffinement d'un service requis. Nous nous intéressons ici à la correction des composants, des assemblages et des composites vis-à-vis des contrats fonctionnels.

Nous ne traitons pas ici du cas (**CF**) relatif à la correction du comportement dynamique d'un service par rapport à l'abstraction sous forme de pré/post-conditions [2].

1. http ://www.atelierb.eu
2. Des expérimentations basées sur l'exécution symbolique (à l'aide de l'outil Key) sont en cours.

En somme, les vérifications à effectuer concernent des propriétés de cohérence : cohérence des services des composants, cohérence des assemblages des composants et de promotion de services.

Dans un premier temps, les spécifications Kmelia sont analysées avec COSTO (vérifications de syntaxe, de type et de cohérence simple). Puis, nous utilisons la méthode B pour mettre en œuvre la démarche de preuve de cohérence. Comme nous l'avons déjà signalé dans la section précédente, la méthode B génère systématiquement des obligations de preuve selon la sémantique de préservation des propriétés invariantes. Ce cadre de preuve est adéquat aux obligations de preuve Kmelia que nous avons proposées dans le chapitre 5. Nous transposons donc la question de la vérification de cohérence de composants Kmelia en une question de vérification de cohérence en B. Dans la suite, nous présentons une vue globale de notre approche (cf. figure 6.1).

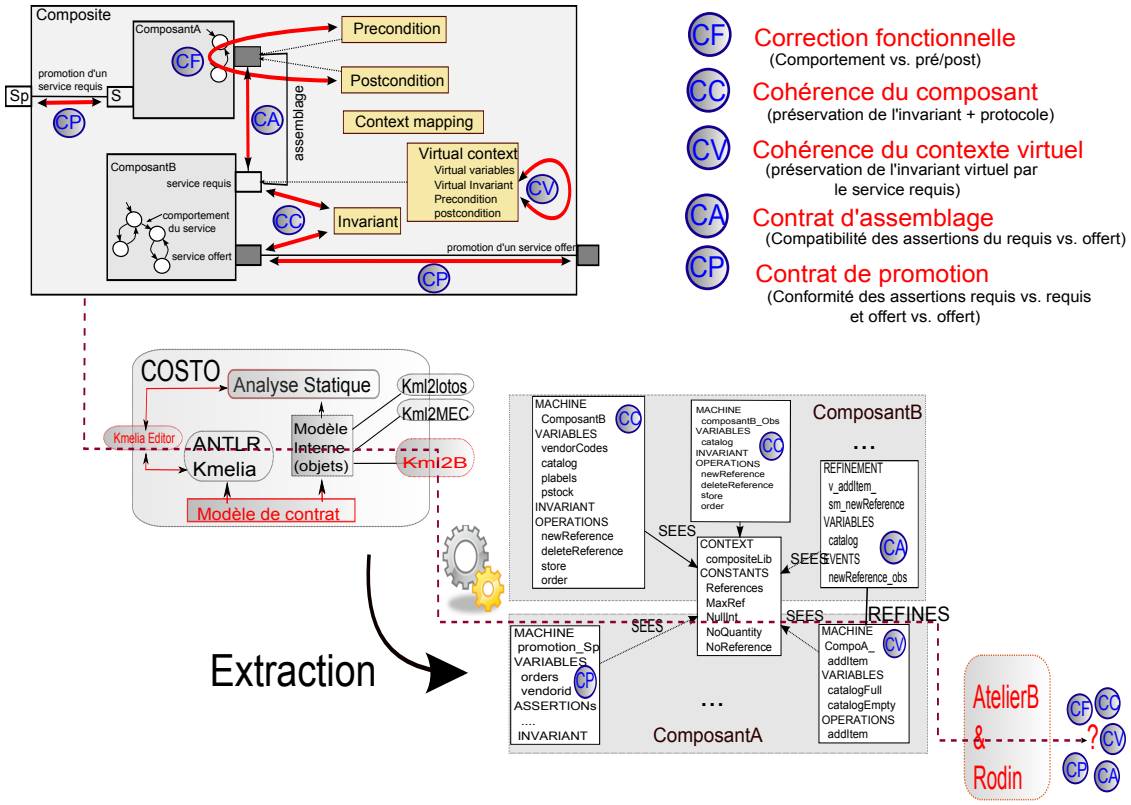


FIGURE 6.1 – Vue globale de notre approche de vérification des contrats fonctionnels.

Afin de vérifier systématiquement les propriétés liées aux contrats, nous proposons de construire des machines B à partir de spécifications Kmelia puis de vérifier ces machines grâce aux outils de B. Nous procédons en trois phases qui couvrent les vérification locales (intra-composant) et globales (inter-composants) :

1. Vérification intra-composant. Pour chaque composant Kmelia on extrait :
   - une machine B pour vérifier sa cohérence (propriété **CC**) ;
   - une autre machine restreinte aux éléments observables du composant pour prouver la cohérence de sa partie observable (propriété **CO**) ;

- autant de machines que de services requis dans le composant, chacune de ses machines permet de vérifier la cohérence du contexte virtuel du service requis correspondant (propriété **CV**).

2. Vérification inter-composants :
   - pour chaque lien d'assemblage, on réutilise la machine extraite à partir du service requis (déjà utilisée pour vérifier la propriété **CV**) et un raffinement à partir du service offert. Les obligations de preuve B de ce raffinement permettent de vérifier la propriété **CA** en Kmelia ;
   - pour chaque lien de promotion, on extrait une machine qui prend ses éléments à la fois du service d'origine et du service promu. Cette machine sert à prouver la conformité entre les contrats des deux services mis en relation par le lien de promotion (propriété **CP**).

Dans la section suivante, nous montrons comment des spécifications B sont extraites des spécifications Kmelia. Des exemples plus détaillés sont également donnés dans le chapitre suivant.

6.3 Extraction de B à partir de Kmelia

Afin de formaliser la traduction du modèle Kmelia, nous avons introduit un morphisme $[\![\ ]\!]_{\mathbb{B}}$ pour définir les règles de transformation d'un arbre de syntaxe abstraite Kmelia en un arbre de syntaxe abstraite B, préservant la sémantique des constructions Kmelia.

Pour chaque composant Kmelia nommé C on extrait une machine C dont l'espace d'état est déduit directement de celui du composant. Les services srv_i dans C sont traduits par des opérations srv_i dans la même machine. Une autre machine C_obs, restreinte aux éléments observables de C, est générée en appliquant le même principe.

6.3.1 Traduction des types Kmelia en B

Les types ***Integer***, ***Boolean***, ***range***, ***enum***, ***struct*** de Kmelia sont directement traduits en B par le morphisme $[\![\ ]\!]_{\mathbb{B}}$. Le type générique ***setOf*** est traduit par l'ensemble des parties noté $\mathbb{P}$ et le type ***array*** est traduit par une fonction totale selon les règles ci-dessous (cf. tableau 6.3.1).

Règles de traduction des types		
$[\![\textbf{\textit{Integer}}]\!]_{\mathbb{B}}$	=	INT
$[\![\textbf{\textit{Boolean}}]\!]_{\mathbb{B}}$	=	BOOL
$[\![\textbf{\textit{range}}\ \text{min..max}]\!]_{\mathbb{B}}$	=	$[\![\text{min}]\!]_{\mathbb{B}}..[\![\text{max}]\!]_{\mathbb{B}}$
$[\![\textbf{\textit{enum}}\{\text{e}_1, ..., \text{e}_n\}]\!]_{\mathbb{B}}$	=	$\{\text{e}_1, ..., \text{e}_n\}$
$[\![\textbf{\textit{struct}}\{\text{c}_1 : \text{type}_1,...,\text{c}_j : \text{type}_j\}]\!]_{\mathbb{B}}$	=	$\text{struct}(\text{c}_1 : [\![\text{type}_1]\!]_{\mathbb{B}},...,\text{c}_j : [\![\text{type}_j]\!]_{\mathbb{B}})$
$[\![\textbf{\textit{setOf}}\ \text{type}]\!]_{\mathbb{B}}$	=	$\mathbb{P}([\![\text{type}]\!]_{\mathbb{B}})$
$[\![\textbf{\textit{array}}[\text{min..max}]\ \text{of}\ \text{type}]\!]_{\mathbb{B}}$	=	$[\![\text{min}]\!]..[\![\text{max}]\!]_{\mathbb{B}} \rightarrow [\![\text{type}]\!]_{\mathbb{B}}$

Tableau 6.3.2 – Règles de traduction des types de données Kmelia en B.

Exemple : considérons la déclaration de la variable pstock suivante (il s'agit d'un morceau du listing 4.3 (cf. page 83).

```
VARIABLES
  pstock : array [Reference] of Integer //product quantity
```

Le type de la variable pstock se traduit en B en appliquant les règles ci-dessus de la façon suivante :

$$\begin{aligned}[\![\text{pstock : : array [1..MaxInt] of setOf Integer}]\!]_{\mathbb{B}} &= \text{pstock} \in [\![1]\!]_{\mathbb{B}}..[\![\text{maxInt}]\!]_{\mathbb{B}} \rightarrow [\![\text{setOf Integer}]\!]_{\mathbb{B}} \\ &= \text{pstock} \in 1..\text{MaxInt} \rightarrow \mathbb{P}([\![\text{Integer}]\!]_{\mathbb{B}}) \\ &= \text{pstock} \in 1..\text{MaxInt} \rightarrow \mathbb{P}(\text{INT})\end{aligned}$$

6.3.2 Traduction des expressions Kmelia en B

Les expressions sont définies sur des opérateurs et des fonctions. Pour les types prédéfinis de Kmelia, la traduction prend en compte la conversion des fonctions prédéfinies sur un type Kmelia en opérateur B ainsi que la traduction des arguments. Par exemple la fonction *union*(s1, s2) qui prend en paramètres deux ensembles s1 et s2 se traduit en B par : s1/s2. En revanche, un traitement spécifique est associé à la fonction prédéfinie old(v) (qui dans une post-condition désigne la valeur d'une variable v avant l'appel du service) basée sur la substitution **ANY** (cf. section 6.3.3).

Règles de traduction des expressions		
$[\![\text{Var\|Cst}]\!]_{\mathbb{B}}$	=	Var|Cst
$[\![\text{KmlUnaryOp (Op, KmlExp)}]\!]_{\mathbb{B}}$	=	BExp ($[\![\text{Op}]\!]_{\mathbb{B}}$, $[\![\text{KmlExp}]\!]$)
$[\![\text{KmlBinOp (Op, KmlExp, KmlExp)}]\!]_{\mathbb{B}}$	=	BExp ($[\![\text{Op}]\!]_{\mathbb{B}}$, $[\![\text{KmlExp}]\!]_{\mathbb{B}}$, $[\![\text{KmlExp}]\!]_{\mathbb{B}}$
$[\![\text{KmlPred(Quantifier,ListExp,KmlExp)}]\!]_{\mathbb{B}}$	=	BPred($[\![\text{Quantifier}]\!]_{\mathbb{B}}$,$[\![\text{ListExp}]\!]_{\mathbb{B}}$,$[\![\text{KmlExp}]\!]_{\mathbb{B}}$)
$[\![\text{KmlFunCall (fname, ListExp)}]\!]_{\mathbb{B}}$	=	BExp (fname, $[\![\text{ListExp}]\!]_{\mathbb{B}}$)
$[\![\text{KmlSetExp (OP, KmlExp)}]\!]_{\mathbb{B}}$	=	$[\![\text{KmlUnaryOp (Op, KmlExp)}]\!]_{\mathbb{B}}$
$[\![\text{KmlSetExp (OP, KmlExp, KmlExp)}]\!]_{\mathbb{B}}$	=	$[\![\text{KmlBinOp (Op, KmlExp, KmlExp)}]\!]_{\mathbb{B}}]$

Tableau 6.3.3 – Règles de traduction des expressions Kmelia en B.

Exemple : prenons l'exemple du prédicat @resultValue [1] suivant :

```
INVARIANT
    @resultValue : (Result <> noReference) implies (notIn(catalog, Result))
```

La traduction de ce prédicat en B est faite de la façon suivante :

1. Il s'agit d'une réécriture simplifiée du prédicat @notreferenced dans l'invariant du listing 4.3 (cf. page 83)

$[\![$(Result <> noReference) *implies* (*notIn*(catalog, Result))$]\!]_{\mathbb{B}}$
$= [\![$KmlBinOp ('implies', KmlBinOp ('<>', Result, noReference), [KmlFunCall ('notIn', ListExp (catalog,Result))]$]\!]_{\mathbb{B}}$
$= \text{BExp}(\text{'}\Rightarrow\text{'}, [\![\text{Result} <> \text{noReference}]\!]_{\mathbb{B}}, [\![\textit{notIn}(\text{catalog}, \text{Result})]\!]_{\mathbb{B}})$
$= (\text{BExp}(\text{'}\neq\text{'}, [\![\text{Result}]\!]_{\mathbb{B}}, [\![\text{noReference}]\!]_{\mathbb{B}})) \Rightarrow (\text{BExp}(\text{'}\notin\text{'}, [\![\text{Result}]\!]_{\mathbb{B}}, [\![\text{catalog}]\!]_{\mathbb{B}}))$
$= (\text{Result} \neq \text{noReference}) \Rightarrow (\text{Result} \notin \text{catalog})$

6.3.3 Traduction de l'espace d'état d'un composant

L'espace d'état $\mathcal{W} = \langle T, V, type, Inv, Init\rangle$ d'un composant C (où $V = V_c \cup V_v$ constantes et variables) est traduit par $[\![\mathcal{W}]\!]_{\mathbb{B}}$ comme suit :

$$[\![\mathcal{W} = \langle T, V, type, Inv\rangle]\!]_{\mathbb{B}} = \begin{cases} \textbf{MACHINE} & C \\ \textbf{SETS} & [\![T]\!]_{\mathbb{B}} \\ \textbf{CONSTANTES} & [\![V_c]\!]_{\mathbb{B}} \\ \textbf{PROPERTIES} & [\![type_c]\!]_{\mathbb{B}} \\ \textbf{VARIABLES} & [\![V_v]\!]_{\mathbb{B}} \\ \textbf{INVARIANT} & [\![type_v]\!]_{\mathbb{B}} \wedge [\![Inv]\!]_{\mathbb{B}} \\ \textbf{INITIALISATION} & [\![Init]\!]_{\mathbb{B}} \\ \text{OPERATIONS} & [\![\mathcal{D}^P]\!]_{\mathbb{B}} \end{cases}$$

où :

- $[\![T]\!]_{\mathbb{B}} = [\![t]\!]_{\mathbb{B}} \mid t \in T$ (cf. section précédente).
- $[\![V]\!]_{\mathbb{B}} = [\![V_c]\!]_{\mathbb{B}} \cup [\![V_v]\!]_{\mathbb{B}} = V_c \cup V_v$ (déclaration des constantes et variables).
  Par exemple, l'espace d'état du composant StockManager présenté dans le listing 4.3 (cf. page 83) est traduit en B comme suit :

```
CONSTANTS
  MaxRef,
  References,
  NoQuantity,
  NoReference

VARIABLES
  vendorCodes,
  catalog, /* obs */
  plabels,
  pstock,
  Result_newreference
```

- $[\![type]\!]_{\mathbb{B}} = \bigwedge [\![x_i]\!]_{\mathbb{B}} \in [\![t_i]\!]_{\mathbb{B}} \mid (x_i, t_i) \in type$ (typage des constantes et variables)
  $= \begin{cases} [\![type_c]\!]_{\mathbb{B}} & \forall (c,t) \in type_c \Rightarrow (c,t) \in type \wedge c \in V_c \\ [\![type_v]\!]_{\mathbb{B}} & \forall (v,t) \in type_v \Rightarrow (v,t) \in type \wedge v \in V_v \end{cases}$
  Par exemple, le typage des constantes (resp. des variables) du composant StockManager (cf. page 83) est déclaré comme une conjonction de prédicats dans la clause **PROPERTIES** (resp. **INVARIANT**) dans la machine B comme suit :

```
PROPERTIES
  MaxRef = 100 ∧
  References = 1..MaxRef ∧
  NoQuantity = −2 ∧
  NoReference = −3
```

```
INVARIANT
  vendorCodes ⊆ ℤ ∧
  catalog ∈ 𝔽(References) ∧ /*obs*/
  plabels ∈ References → String ∧
  pstock ∈ References → ℤ ∧
  Result_newreference ∈ INT ∧
```

– $[\![Inv]\!]_{\mathbb{B}} = \forall pred \in Inv \bullet \bigwedge [\![pred]\!]_{\mathbb{B}}$ (prédicats de l'invariant)
Par exemple, l'invariant du composant StockManager (cf. page 83) complète le typage des variables dans la clause **INVARIANT** dans la machine B et se traduit de la façon suivante :

```
  /*@borned*/ card(catalog) ≤ MaxRef    /*obs*/
  ∧ /*@referenced*/ ∀ ref1 . (ref1 ∈ References ∧ ref1 ∈ catalog ⇒
        ( plabels( ref1 ) ≠ EmptyString ∧ pstock(ref1) ≠ NoQuantity))
  ∧ /*@notreferenced*/ ∀ ref2 . (( ref2 ∈ References ∧ ref2 /∈ catalog) ⇒
            (( plabels( ref2 ) = EmptyString ∧ pstock(ref2) = NoQuantity)))
```

– $[\![Init]\!]_{\mathbb{B}} = \{([\![v]\!]_{\mathbb{B}}, [\![expr]\!]_{\mathbb{B}}) \mid (v, expr) \in Init\}$ (état initial)
Par exemple, l'initialisation du composant StockManager (cf. page 83) se traduit par une substitution généralisée dans la clause **INITIALISATION** comme suit :

```
INITIALISATION
vendorCodes := ∅ ||
catalog := ∅ ||
plabels := (1..MaxRef) × {EmptyString} ||
pstock := (1..MaxRef) × {NoQuantity}||
Result_newreference := 0
```

6.3.4 Traduction des services Kmelia en B

L'ensemble $\mathcal{D}^P$ des services offerts d'un composant C est traduit par un ensemble d'opérations . $[\![\mathcal{D}^P]\!]_{\mathbb{B}} = \{[\![srv]\!]_{\mathbb{B}} \mid srv \in \mathcal{D}^P\}$. Un service est décrit formellement par un triplet $\langle \mathcal{IS}, l\mathcal{W}, \mathcal{B} \rangle$. Dans le contexte de la vérification de pré/post-conditions des services par rapport à l'invariant du composant, seule l'interface $\mathcal{IS} = \langle \sigma, Pre, Post \rangle$ est concernée par la traduction [1]. Le schéma de traduction des services est le suivant :

1. L'espace d'état $l\mathcal{W}$ est exploité ultérieurement pour vérifier les liens d'assemblages. Le comportement $\mathcal{B}$ est utilisé pour la vérification fonctionnelle du service lui-même.

$$[\![srv]\!]_{\mathbb{B}} = [\![\langle \sigma, Pre, Post \rangle]\!]_{\mathbb{B}} = \left\{ \begin{array}{ll} [\![\sigma]\!]_{\mathbb{B}} = \left\{ \begin{array}{l} [\![name]\!]_{\mathbb{B}} \\ [\![param]\!]_{\mathbb{B}} \\ [\![ptype]\!]_{\mathbb{B}} \\ [\![Tres]\!]_{\mathbb{B}} \end{array} \right. & = \{\ result \leftarrow name([\![param]\!]_{\mathbb{B}}) \\ [\![Pre]\!]_{\mathbb{B}} & = \left\{ \begin{array}{l} \textbf{PRE} \\ \quad [\![ptype]\!]_{\mathbb{B}} \wedge \\ \quad [\![Pre]\!]_{\mathbb{B}} \end{array} \right. \\ [\![Post]\!]_{\mathbb{B}} & = \left\{ \begin{array}{l} \textbf{ANY} \\ \quad [\![\alpha_l\_V_{Post}]\!]_{\mathbb{B}} \cup \{l\_result\} \\ \textbf{WHERE} \\ \quad [\![\alpha_l\_Post]\!]_{\mathbb{B}} \wedge \\ \quad \alpha_l\_type(V_{Post}) \wedge \\ \quad l\_result \in [\![Tres]\!]_{\mathbb{B}} \\ \textbf{THEN} \\ \quad [\![V_{Post}]\!]_{\mathbb{B}} := [\![\alpha_l\_V_{Post}]\!]_{\mathbb{B}} \ || \\ \quad result := l\_result \ || \\ \quad name\_res := l\_result \end{array} \right. \end{array} \right.$$

où :

- *result* est la valeur de retour du service, *name* est le nom du service et $[\![param]\!]_{\mathbb{B}} = param$ est l'ensemble des paramètres. Ces trois éléments constituent la signature de l'opération.
  Par exemple, la signature du service newReference du composant StockManager présenté dans le listing 4.4 (cf. page 84) est traduite en B comme suit :

```
Result ⟵ newReference =
```

- la pré-condition de l'opération B assure le typage des paramètres en plus de la pré-condition du service :
  - $[\![ptype]\!]_{\mathbb{B}} = \bigwedge [\![p_i]\!]_{\mathbb{B}} \in [\![t_i]\!]_{\mathbb{B}} \mid (p_i, t_i) \in ptype \wedge [\![pre]\!]_{\mathbb{B}} = \forall pred \in Pre \bullet \bigwedge [\![pred]\!]_{\mathbb{B}}$.

  Par exemple, la pré-condition du service newReference (cf. page 84) est traduite en B dans la clause **PRE** comme suit :

```
PRE
    card(catalog) < MaxRef
```

- une post-condition Kmelia établit une relation entre les variables avant et après l'exécution. Elle est traduite par une substitution non déterministe[1] sur des variables locales (clause **ANY**) de même type que les variables modifiées et qui satisfont la post-condition (clause **WHERE**). Ces variables locales sont affectées aux variables d'état de la machine (clause **THEN**).
  - soit α_l_θ une α-conversion qui remplace les occurrences d'une variable v de θ par l_v.
  - $[\![\alpha_l\_V_{Post}]\!]_{\mathbb{B}}$ est un ensemble de variables locales qui représentent les valeurs des variables de la machine après l'exécution de l'opération. Seules les variables modifiées sont concernées.

1. Nous aurions pu utiliser le prédicat *before-after* de Event-B, mais dans la version actuelle de l'extraction, on génère du B classique.

Par exemple, les variables l_Result, l_catalog, l_pstock et l_plabels sont introduites dans la clause **ANY** pour modéliser les variables catalog, pstock et plabels du composant StockManager et la variable Result du service newReference après l'exécution de l'opération newReference.

```
BEGIN
    ANY l_Result, l_catalog, l_pstock, l_plabels
```

– $\{l\_result\}$ est une variable locale modélisant la valeur de retour.

– $[\![\alpha_l\_type]\!]_{\mathbb{B}} = \forall(v,t) \in type \bullet \bigwedge [\![l\_v]\!]_{\mathbb{B}} \in [\![t]\!]_{\mathbb{B}}$.

Par exemple, toutes les variables locales introduites dans la clause **ANY** ci-dessus, doivent être typées dans la clause **WHERE** de la façon suivante :

```
WHERE
      l_Result ∈ ℤ ∧
      l_catalog ∈ 𝔽(References) ∧
      l_plabels ∈ References → String ∧
      l_pstock ∈ References → ℤ
```

– $[\![\alpha_l\_Post]\!]_{\mathbb{B}} = \forall pred \in Post \bullet \bigwedge [\![\alpha_l\_pred]\!]_{\mathbb{B}}$.

Par exemple, la post-condition du service newReference (cf. page 84) est traduite en B dans la clause **WHERE** de la façon suivante :

```
∧
( (l_Result > 0 ∧ l_Result ≤ MaxRef) ∨ l_Result = NoReference ) /*obs*/
∧ (l_Result ≠ NoReference ⇒ ( l_catalog = catalog ∪ {l_Result} ) ) /*obs*/
∧ (l_Result = NoReference ⇒ l_catalog = catalog ) /*obs*/
∧ (l_Result ≠ NoReference ⇒ (l_pstock(l_Result) = 0 ∧ l_plabels(l_Result)∈String−{EmptyString}
                        ∧ ∀ ii . ( ii ∈ 1..MaxRef ∧ ii ≠ l_Result ⇒(l_pstock(ii) = pstock(ii)
                        ∧ l_plabels(ii) = plabels(ii) ))))
∧ (l_Result = NoReference ⇒ ( ∀ ii . (ii ∈ 1..MaxRef ⇒
                          (l_pstock(ii) = pstock(ii) ∧ l_plabels(ii) = plabels(ii) )) ))/*obs*/
```

Enfin, les variables locales sont affectées aux variables d'état de la machine comme le montre l'exemple suivant :

```
THEN
    Result := l_Result ||
    Result_newreference := l_Result ||
    catalog := l_catalog ||
    pstock := l_pstock ||
    plabels := l_plabels
END
```

– *name_res* représente la variable modélisant la valeur de retour du service *name* dans le composant (Result_newreference dans l'exemple ci-dessus).

6.4 Vérifications intra-composant

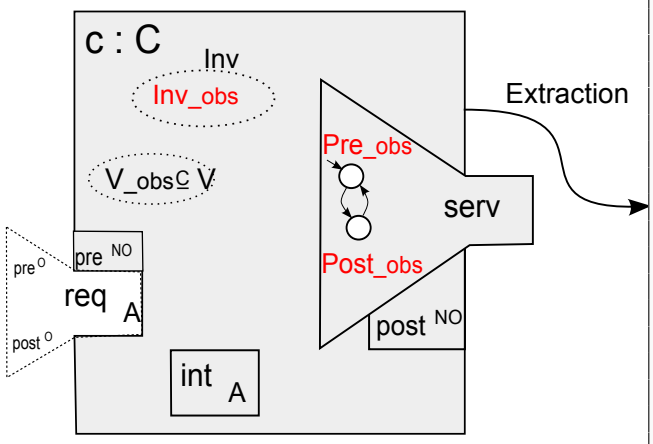
Dans cette section, nous montrons comment nous avons tiré parti du processus de traduction présenté précédemment. Dans un premier temps, nous nous intéressons à la cohérence des composants individuels. Nous présentons ainsi les différentes

machines B à extraire afin de vérifier les deux premières obligations de preuve établies dans la section 5.3.2. La troisième sera traitée dans le cadre de la vérification des contrats d'assemblage dans la section 6.5.1.1.

6.4.1 Vérification de la cohérence de la partie observable du composant

Pour chaque composant Kmelia nous procédons ainsi :

1. Nous gênerons d'abord une machine B comme illustré dans le *pattern* du listing 6.3. Les variables observables et l'invariant du composant Kmelia deviennent les variables et l'invariant de la machine C_obs.



Listing 6.3 – *Pattern* C_obs

```
MACHINE
    C_obs
VARIABLES
    V_obs, res
INVARIANT
    [[V_obs ∈ T_obs]]B ∧
    [[Inv_obs]]B ∧
    [[res ∈ T_res]]B
INITIALISATION
    [[ Init ]]B
OPERATIONS
    serv_obs (param) ≙
        PRE
            [[Pre(p,V_obs)]]B ∧
            param ∈ T_param
        ANY I_V_obs
        WHERE
            [[Post(V_obs)]]B
        THEN
            V_obs, res := I_V_obs, I_res
        END
END
```

FIGURE 6.2 – Principe d'extraction d'une machine B pour vérifier la cohérence de la partie observable d'un composant.

2. Nous traduisons chaque service offert dans l'interface du composant par une opération B. Les paramètres de cette dernière sont les paramètres du service et la pré-condition du service Kmelia devient une pré-condition de l'opération B.
3. Nous modélisons le corps de l'opération par une substitution généralisée issue de la traduction de la partie observable de la post-condition (cf. section 6.3).
4. Enfin, nous utilisons le *pattern* C_obs pour vérifier la cohérence de la partie observable des composants Kmelia.

Cependant, il faut montrer que les preuves faites en B sur la machine C_obs garantissent l'obligation de preuve **CO** de Kmelia. Pour ce faire, il faut prouver l'équivalence entre ces deux obligations comme nous le détaillons dans la suite.

Proposition 6.1 (Cohérence : CO) *La preuve de la cohérence de la machine C_ obs générée assure la preuve de la propriété CO concernant la cohérence de la partie observable du composant, c'est-à-dire que la partie observable du contrat du service est suffisante pour établir la partie observable de l'invariant (cf. section 5.3.2).*

$$\boxed{Inv^O(before) \wedge Pre^O(before) \wedge Post^O(before, after) \Rightarrow Inv^O(before, after)} \quad \textbf{(CO)}$$

Où : $before$ (resp. $after$) fait référence aux valeurs de l'espace d'état du composant juste avant l'appel du service (resp. juste après la terminaison du service) et l'indice ^{O} fait référence aux éléments observables (cf. section 4.4.1).

Preuve 6.1 *L'obligation de preuve associée à l'opération serv_ obs de la machine C_ obs est :*

$$\boxed{P \wedge J \wedge I \wedge Pre \Rightarrow [S]I} \text{ (cf. définition 6.3).}$$

*où P est le prédicat modélisant les propriétés, J une conjonction d'assertions, I l'invariant de la machine, Pre la pré-condition de l'opération, $[S]I$ est l'ensemble des substitutions de l'opération serv_ obs appliquées au prédicat de l'invariant I. Notons que la machine ne contient ni la clause **PROPERTIES** ni la clause **ASSERTIONS**, donc les prédicats P et J sont réduits à TRUE. On obtient ainsi :*

$$\boxed{I \wedge Pre \Rightarrow [S]I}$$

Appliquée à la machine C_ obs, l'obligation s'écrit alors :

$\equiv [\![V\_obs \in T\_obs]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![res \in T\_res]\!]_{\mathbb{B}} \wedge [\![Pre(p, V\_obs)]\!]_{\mathbb{B}}$
$\wedge\ param \in T\_param \Rightarrow [\ \textbf{ANY}\ l\_V\_obs\ \textbf{WHERE}\ [\![Post(V\_obs)]\!]_{\mathbb{B}}\ \textbf{THEN}$
$V\_obs,\ res := l\_V\_obs,\ l\_res\ \textbf{END}]I$

$\equiv [\![V\_obs \in T\_obs]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![res \in T\_res]\!]_{\mathbb{B}} \wedge [\![Pre(p, V\_obs)]\!]_{\mathbb{B}}$
$\wedge\ param \in T\_param \Rightarrow (l\_V\_obs : [\![Post(V\_obs)]\!]_{\mathbb{B}}$
$\Rightarrow [V\_obs, res := l\_V\_obs, l\_res]I)$

$\equiv [\![V\_obs \in T\_obs]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![res \in T\_res]\!]_{\mathbb{B}} \wedge [\![Pre(p, V\_obs)]\!]_{\mathbb{B}}$
$\wedge\ param \in T\_param \wedge l\_V\_obs : [\![Post(V\_obs)]\!]_{\mathbb{B}} \Rightarrow [V\_obs, res := l\_V\_obs, l\_res]I$

$\equiv [\![V\_obs \in T\_obs]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![res \in T\_res]\!]_{\mathbb{B}} \wedge [\![Pre(p, V\_obs)]\!]_{\mathbb{B}}$
$\wedge\ param \in T\_param \wedge [\![Post(V\_obs,\ l\_V\_obs)]\!]_{\mathbb{B}} \Rightarrow I\ (V\_obs,\ l\_V\_obs)$

Par rétroaction sur les éléments Kmelia on obtient :

$\equiv [\![[\![V\_obs \in T\_obs]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![res \in T\_res]\!]_{\mathbb{B}} \wedge [\![Pre(p, V\_obs)]\!]_{\mathbb{B}}$
$\wedge\ param \in T\_param \wedge [\![Post(V\_obs,\ l\_V\_obs)]\!]_{\mathbb{B}}]\!]_{\mathbb{B}^{-1}} \Rightarrow [\![I\ (V\_obs,\ l\_V\_obs)]\!]_{\mathbb{B}^{-1}}$

$\equiv Inv\_obs \wedge Pre(p, V\_obs) \wedge Post(V\_obs,\ l\_V\_obs) \Rightarrow Inv\_obs(V\_obs, l\_V\_obs)$

Ce qui établit la proposition 6.1. □

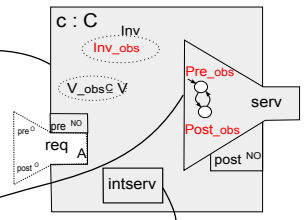
Les prédicats $[\![V\_obs \in T\_obs]\!]_{\mathbb{B}^{-1}} \wedge [\![res \in T\_res]\!]_{\mathbb{B}^{-1}} \wedge param \in T\_param$ ont disparu au niveau Kmelia car, du point de vue de la traçabilité, ils ne proviennent pas des éléments de contrats fonctionnels (cf. section 6.3.3 et 6.3.4), mais font plutôt partie des contrats structurels (le typage Kmelia se faisant en même temps que la déclaration des variables).

6.4.2 Vérification de la cohérence du composant

Maintenant, nous considérons tout le composant et non pas uniquement sa partie observable. Nous construisons une autre machine B que nous appelons (C). Cette dernière reprend les mêmes éléments que la machine C_obs auparavant générée (cf. section précédente) en y ajoutant les variables non-observables et la partie non-observable de l'invariant. Pour chaque service offert, on ajoute la traduction des post-conditions locales dans le corps de l'opération correspondante. Chaque service interne est également traduit par une opération B. Le *pattern* 6.4 donne un aperçu du schéma de traduction.

Listing 6.4 – Pattern C

```
MACHINE
    C
VARIABLES
    V, res      /* V = V_obs ∪ V_nobs*/
INVARIANT
    V ∈ T ∧
    Inv         /* Inv = Inv_obs ∧ Inv_nobs*/
INITIALISATION
    [[ Init ]]B
OPERATIONS
 serv (param) ≙          /* service offert */
    PRE
        Pre(V_obs) ∧
        param ∈ T_param
    ANY
        I_V, I_res
    WHERE
        Post(I_V, I_res)
    THEN
        V, res := I_V, I_res
    END
```




```
intserv ≙          /* service interne */
    PRE
        iPre(V) ∧
        param ∈ T_param
    ANY
        I_V, I_res
    WHERE
        Post(I_V, I_res)
    THEN
        V, res := I_V, I_res
    END
END
```

FIGURE 6.3 – Principe d'extraction d'une machine B pour vérifier la cohérence globale d'un composant.

Nous utilisons le *pattern* C pour vérifier la cohérence d'un composant Kmelia. L'obligation de preuve que nous avons établie dans la section 5.3.2 est la suivante :

$$Inv^O(before) \wedge Inv(before) \wedge Pre^O(before) \wedge Post^O(before, after) \wedge Post^I \Rightarrow Inv^O \wedge Inv \qquad \textbf{(CC)}$$

Proposition 6.2 (Cohérence : CC) *La preuve de la machine C permet la vérification des propriétés CC pour la préservation de l'invariant par le composant (y compris la partie non-observable).*

Preuve 6.2 *L'obligation de preuve B concernant l'opération serv (resp. intserv) de la machine C correspond exactement à la règle CC. Pour démontrer cette proposition, nous procédons de la même manière que dans la preuve 6.1.* □

6.5 Vérifications inter-composants

Cette section expose la manière dont nous avons utilisé le processus de traduction de la section 6.3 dans le cadre de la vérification des contrats d'assemblage et de promotion. L'objectif étant de générer des spécifications B de telle sorte que les obligations de preuve générées soient équivalentes à celles que nous avons définies dans la section 5.4.

6.5.1 Vérification du contrat d'assemblage

Dans cette section, nous nous intéressons aux liens d'assemblage entre les services requis et les services offerts au sein d'un assemblage de composants. Soit $(C_p, servP, C_r, servR)$ un lien d'assemblage qui relie un service requis $servR$ d'un composant C_r à un service offert $servP$ d'un autre composant C_p (cf. section 3.5.2.1).

Pour transposer la correction du contrat d'assemblage en B, notre principe est de considérer le service offert $servP$ comme un raffinement du service requis $servR$ de telle sorte que les obligations de preuve générées pour la correction de ce dernier soient équivalentes à celles que nous avons définies au niveau Kmelia (**CA**).

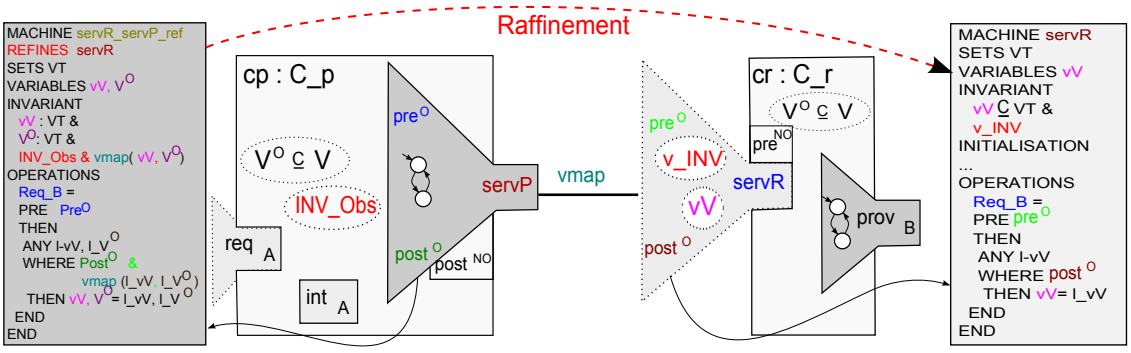


FIGURE 6.4 – Principe d'extraction de spécifications B à partir d'un assemblage.

La mise en œuvre de cette solution nécessite la génération d'une machine sr à partir du service requis $servR$ (l'espace d'état virtuel) et un raffinement sr_sp_ref à partir du service offert $servP$ qui raffine la machine sr (cf. figure 6.4). Nous détaillons ces étapes dans les deux sections suivantes.

6.5.1.1 Extraction d'une machine B à partir du service requis

Pour chaque service requis, une machine B doit être générée. Le contexte virtuel d'un service requis au niveau Kmelia devient l'espace d'état de la machine B, c'est-à-dire que les variables virtuelles et l'invariant virtuel deviennent respectivement les variables et l'invariant de la machine (selon les règles de traduction définies précédemment). Cependant, le service requis se traduit par une seule opération comme indiqué dans le listing 6.5.

Listing 6.5 – Pattern C_servR

```
MACHINE sr
VARIABLES
    V_sr, rs
INVARIANT
    [[type(V_sr)]]_B ∧ [[Inv_sr]]_B ∧ rs ∈ [[T_rs]]_B
INITIALISATION
    [[ Init ]]_B
OPERATIONS
  rr ⟵ servR (P) =
    Pre
      [[type(P)]]_B ∧ [[Pre_sr]]_B
    ANY
      α_l_V_sr, l_rs
    WHERE
      [[α_l_Post_sr]]_B ∧ [[α_l_type(V_sr)]]_B ∧
      l_r ∈ [[T_res]]_B
    THEN
      V_sr := α_l_V_sr || rs := l_rs
    END
END
```

Cette machine est utilisée à la fois dans les vérifications locales et dans les vérifications globales. Au niveau local, cette machine permet de vérifier la cohérence du service requis exprimé par l'obligation de preuve suivante :

$$\boxed{Inv^V(before) \wedge Pre^V(before) \wedge Post^V(before, after) \Rightarrow Inv^V} \quad \textbf{(CV)}$$

Où l'indice ^{V} fait référence aux éléments du contexte virtuel du service requis.

Proposition 6.3 (Cohérence : CV) *Les obligations de preuve B générées pour la machine C_servR assurent la preuve de la règle* `CV` *pour la cohérence de l'invariant du service requis servR.*

Preuve 6.3 *La correspondance entre l'obligation de preuve générée concernant la préservation de l'invariant par l'opération C_servR et l'obligation de preuve Kmelia* ***CV*** *peut être démontrée en suivant la même démarche que dans la preuve 6.1.* □

Au niveau global, la machine C_servR est utilisée pour vérifier le contrat d'assemblage comme nous allons le détailler dans la section suivante.

6.5.1.2 Extraction d'un raffinement B à partir du service offert

Le raffinement servR_servP_ref raffine la machine servR en y ajoutant les variables observables du service $servP$. L'invariant est complété par le prédicat de collage Map qui exprime le *mapping* entre les variables virtuelles du service requis $servR$ et les variables observables du service offert $servP$;
- $[\![Map(\alpha_l\_V_{sr}, Exp(\alpha_l\_V_{obs}))]\!]_{\mathbb{B}} = \forall v, E(V_{obs}) \in Map \bullet \bigwedge l\_v = [\![E(\alpha_l\_V_{obs})]\!]_{\mathbb{B}}$, où $E(V_{obs})$ est une expression Kmelia sur les variables observables de $servP$.

Listing 6.6 – Pattern C_ref

```
REFINEMENT sr_sp_ref
REFINES sr
VARIABLES
    V_sr, rp, V_obs
INVARIANT
    [[type(V_obs)]]_B ∧ [[Inv_obs]]_B ∧
    [[Map(V_sr, Exp(V_obs))]]_B ∧ rp ∈ [[T_rp]]_B
INITIALISATION
    [[ Init ]]_B
OPERATIONS
    rr ⟵ servR (P) =
        Pre
            [[type(P)]]_B ∧ [[Pre_obs]]_B
        ANY
            α_l_V_obs, α_l_V_sr, l_rp
        WHERE
            [[α_l_Post_sr]]_B ∧ [[Map (α_l_V_sr, Exp(α_l_V_obs))]]_B
            [[α_l_type(V_sr)]]_B ∧ l_r ∈ [[Tres]]_B
        THEN
            V_sr, V_obs, rp := α_l_V_sr, α_l_V_obs, l_rp
        END
END
```

Où [[Map(V_sr, Exp(V_obs))]]$_{\mathbb{B}}$ est le prédicat de liaison (cf. section 4.4.2) permettant de lier les variables du contexte virtuel du service requis et les variables de l'espace d'état du composant du service offert. Dans la suite, nous utilisons ce raffinement pour prouver la correction du contrat d'assemblage.

6.5.1.3 Vérification

Le contrat qui établit un assemblage correct en Kmelia stipule que la pré-condition de $servR$ est plus forte que celle de $servP$, et que la post-condition de $servR$ est, elle, plus faible que celle de $servP$ (cf. section 5.4.1) :

$$\boxed{Inv^V \wedge Inv^O \wedge Map \wedge Pre^V_{cm} \Rightarrow Pre^O}$$

$$\boxed{Inv^O(before) \wedge Post^O(befor, after) \Rightarrow (Pre^V_{cm}(before) \Rightarrow Post^V_{cm}(before, after))} \quad \textbf{(CA)}$$

De la même manière que dans la section 6.4.1, dans la suite, nous détaillons la démonstration de l'équivalence entre l'obligation de preuve B et celle de Kmelia.

Proposition 6.4 (Contrat d'assemblage : CA) *La preuve du raffinement entre les deux opérations B (serv_R et serv_P) permet de démontrer l'obligation de preuve liée au contrat d'assemblage (**CA**) en Kmelia que nous avons établie dans la proposition 5.1.*

Preuve 6.4 *L'obligation de preuve ci-dessous concerne la preuve d'un raffinement d'une opération serv_ R par une opération serv_ P (cf. définition 6.4) :*

$$\boxed{P \wedge J \wedge I_R \wedge Pre_R \wedge I_P \Rightarrow Pre_P \wedge [S_P]\neg[S_R]\neg I_P}$$

*Où l'indice $_R$ (resp. $_P$) fait référence aux éléments de la machine abstraite générée à partir du service requis servR (resp. du raffinement généré à partir du service offert servP) et l'on suppose que les opérations ont la forme générique **PRE** P **THEN** S **END** pour alléger la présentation de la preuve. Comme dans la preuve 6.1, les prédicats P et J sont réduits à TRUE car le raffinement ne contient pas d'assertions ni de propriétés de constantes. On obtient ainsi :*

$$\boxed{I_R \wedge Pre_R \wedge I_P \Rightarrow Pre_P \wedge [S_P]\neg[S_R]\neg I_P}$$

On réécrit l'obligation sous la forme suivante :

$$\equiv \begin{cases} 1.\ I_R \wedge Pre_R \wedge I_P \Rightarrow Pre_P \\ 2.\ I_R \wedge Pre_R \wedge I_P \Rightarrow [S_P]\neg[S_R]\neg I_P \end{cases}$$

1. *I_P est l'invariant de liaison défini au niveau du raffinement. Dans le* pattern *C_ref cet invariant est modélisé par une conjonction de l'invariant du composant offrant le service servP ($[\![Inv\_obs]\!]_{\mathbb{B}}$) et du prédicat de* mapping *$[\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}}$. L'obligation* 1. *s'écrit alors :*

   $\equiv [\![Inv\_sr]\!]_{\mathbb{B}} \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}} \Rightarrow [\![Pre\_obs]\!]_{\mathbb{B}}$

   Par rétroaction sur les éléments Kmelia on obtient :

   $\equiv [\![[\![Inv\_sr]\!]_{\mathbb{B}} \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}}]\!]_{\mathbb{B}^{-1}} \Rightarrow [\![[\![Pre\_obs]\!]_{\mathbb{B}}]\!]_{\mathbb{B}^{-1}}$

   $\equiv Inv\_sr \wedge Inv\_obs \wedge Pre\_sr \wedge Map(V\_sr, Exp(V\_obs)) \Rightarrow Pre\_obs$

   Ce qui établit l'obligation 1 *(première partie de l'obligation **CA**).*

2. *La deuxième obligation de preuve concerne le raffinement du corps de l'opération. Appliquée à l'opération servR du* pattern *C_ref, elle se réécrit ainsi :*

   $\equiv [\![Inv\_sr]\!]_{\mathbb{B}} \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}} \Rightarrow [S_P]\neg[S_R]\neg([\![Inv\_sr]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}})$

   où S_P (resp. S_R) est le corps de l'opération servR au niveau du raffinement C_ref (resp. de la machine abstraite C_servR). Afin d'alléger la présentation de la preuve, notons aussi que $I_P =$ $[\![Inv\_sr]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}}$, on obtient donc :

   $\equiv I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \Rightarrow [S_P]\neg[S_R]\neg I_P$

   $\equiv I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \Rightarrow [V_{obs} :[\![Post_P]\!]_{\mathbb{B}}]\neg[V_{sr} :[\![Post_R]\!]_{\mathbb{B}}]\neg I_P$

   $\equiv I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \Rightarrow \forall l\_V_{obs}([\![Post_P]\!]_{\mathbb{B}} \Rightarrow [V_{obs} := l\_V_{obs}])$
   $\neg(\forall l\_V_{sr}([\![Post_R]\!]_{\mathbb{B}} \Rightarrow [V_{sr} := l\_V_{sr}])\neg I_P)$

$\equiv I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \Rightarrow \forall l\_V_{obs}([\![Post_P]\!]_{\mathbb{B}} \Rightarrow [V_{obs} := l\_V_{obs}])$
$(\exists l\_V_{sr}([\![Post_R]\!]_{\mathbb{B}} \Rightarrow [V_{sr} := l\_V_{sr}])I_P)$

$\equiv \forall l\_V_{obs}(I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}}$
$\wedge [\![Post_P]\!]_{\mathbb{B}}(V_{obs}, l\_V_{obs}) \Rightarrow \exists l\_V_{sr}([\![Post_R]\!]_{\mathbb{B}}(V_{sr}, l\_V_{sr}) \wedge I_P))$ *(cf. règles DR 8 DR 11 pages 758 et 759 dans [Abr96])*

$\equiv I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}}$
$\wedge [\![Post_P]\!]_{\mathbb{B}}(V_{obs}, l\_V_{obs}) \Rightarrow [\![Post_R]\!]_{\mathbb{B}}(V_{sr}, l\_V_{sr}) \wedge I_P(l\_V_{sr}, l\_V_{obs})$

car $I_P = [\![Inv\_sr]\!]_{\mathbb{B}} \wedge [\![Inv\_obs]\!]_{\mathbb{B}} \wedge [\![Map(V\_sr, Exp(V\_obs))]\!]_{\mathbb{B}}$, *et nous avons aussi* $[V_{sr} := l\_V_{sr}]$ *et* $[V_{obs} := l\_V_{obs}]$. *On obtient donc* $[\![Map(l\_V\_sr, Exp(Vl\_V\_obs))]\!]_{\mathbb{B}}$

$\equiv I_P \wedge [\![Pre\_sr]\!]_{\mathbb{B}} \wedge [\![Post_P]\!]_{\mathbb{B}}(V_{obs}, l\_V_{obs}) \Rightarrow [\![Post_R]\!]_{\mathbb{B}}(V_{sr}, l\_V_{sr})$

$\equiv I_P \wedge [\![Post_P]\!]_{\mathbb{B}}(V_{obs}, l\_V_{obs}) \Rightarrow ([\![Pre\_sr]\!]_{\mathbb{B}} \Rightarrow [\![Post_R]\!]_{\mathbb{B}}(V_{sr}, l\_V_{sr}))$

Par rétroaction sur les éléments Kmelia on obtient :

$\equiv [\![I_P]\!]_{\mathbb{B}^{-1}} \wedge [\![[\![Post_P]\!]_{\mathbb{B}}(V_{obs}, l\_V_{obs})]\!]_{\mathbb{B}^{-1}} \Rightarrow ([\![[\![Pre\_sr]\!]_{\mathbb{B}}]\!]_{\mathbb{B}^{-1}}$
$\Rightarrow [\![[\![Post_R]\!]_{\mathbb{B}}(V_{sr}, l\_V_{sr}))]\!]_{\mathbb{B}^{-1}}$

$\equiv I_P \wedge Post_P(V_{obs}) \Rightarrow (Pre_R \Rightarrow Post_R(V_{sr}, l\_V_{sr}))$

Ce qui établit l'obligation 2 (deuxième partie de l'obligation ***CA****).* □

6.5.2 Vérification du contrat de promotion

Dans cette section, nous proposons une solution pour vérifier la correction des contrats de promotion. Nous considérons un lien de promotion entre le service offert origin du composant CA et le service promoted du composite CC (cf. figure 6.5).

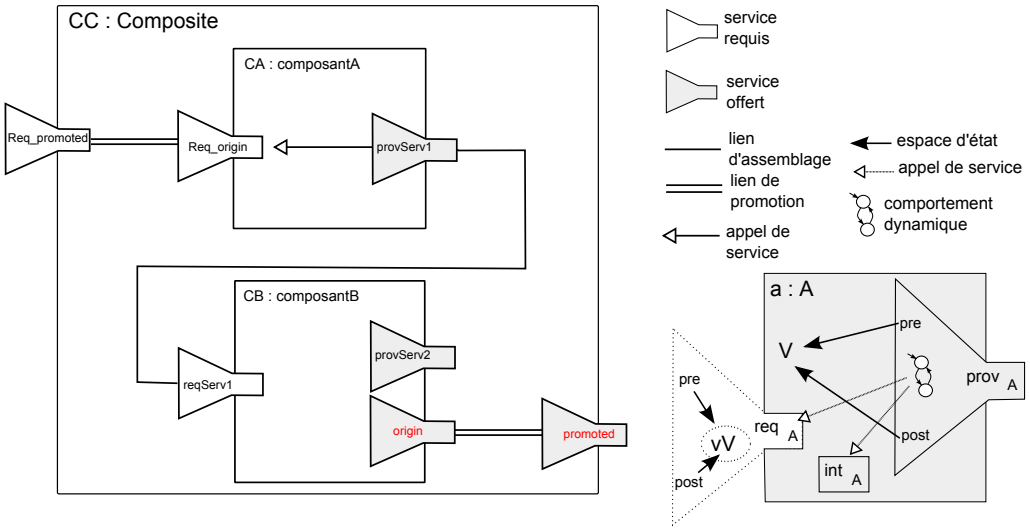


FIGURE 6.5 – Promotion au niveau d'un composite

Nous rappelons que le contrat du service origin garantit que si la pré-condition Pre_{origin} du service origin et l'invariant du composant CA sont vrais, alors le service

origin satisfait sa post-condition $Post_{origin}$. Ce contrat peut être exprimé comme suit :

$$\boxed{Pre_{origin}(before) \wedge inv_C(before) \Rightarrow Post_{origin}(after) \wedge inv_C(after)}$$

6.5.2.1 Extraction d'une machine B à partir d'une promotion de service

Notre solution pour vérifier la correction de la promotion des services est la suivante : nous générons une machine B à partir des deux services (celui d'origine et le promu), de manière à ce que la preuve de cohérence de cette machine établisse la correction du service promu (contrat du service). À ce stade, les contrats de clientèle des sous-composants sont supposés être prouvés comme étant corrects.

Listing 6.7 – Un *pattern* B pour la vérification de la promotion d'un service

```
MACHINE
   C_CC_origin_promoted
CONSTANTS
   /* ORIGINAL */
    /* C variables */
      C_v1, ...
    /* C variables updated values */
      C_v1_new, ...
    /* parameters of origin service */
      o_param1,... , o_result,
   /* PROMOTION */
    /* C variables promoted in CC */
      CC_v1, ...
    /* C variables promoted in CC, updated values */
      CC_v1_new, ...
    /* parameters of promoted service */
       p_param1, ..., p_result
PROPERTIES
   /* ORIGINAL */
    /* C variables and origin parameters typing */
       C_v1∈T ∧ o_param1∈T ∧
    /* C invariant related to the above C variables and their updated values */
       [[Inv_C]]_B ∧ [[Inv_C_new]]_B
    /* post−condition of service origin */
       [[Post_origin]]_B ∧
  /* PROMOTION */
   /* C variables promoted in CC and promoted parameters typing */
       CC_v1∈T ∧ p_param1∈T ∧
   /* promotion variable mapping */
       CC_v1=C_v1 ∧
   /* parameter mapping */
       p_result=o_result ∧
   /* precondition of promoted service − hypothese for the client contract */
       [[Pre_promoted]]_B ∧
ASSERTIONS
   /*precondition of the origin service */
      [[Pre_origin]]_B ∧
   /*post−condition of the promoted service */
      [[Post_promoted]]_B
END
```

La clause **CONSTANTS** de la machine B contient des variables C_v1,... du composant de base C, les paramètres o_param1,... du service *origin* et une variable

modélisant le résultat du service o_result. Elle contient également les mêmes éléments pour le service promu, cette fois-ci préfixés par CC_ pour les variables et p_ pour les paramètres et la variable de retour du service promu.

La clause **PROPERTIES** contient les informations sur le typage de chaque variable et paramètre impliqués dans la machine (C_v1∈T ∧ o_param1∈T ∧ ...), des prédicats de mapping entre les variables d'origine et les variables promues (CC_v1=C_v1 ∧ ...). De plus les prédicats ci-dessous sont utilisés comme des axiomes :

- la pré-condition promue Pre_promoted qui est supposée satisfaite par le client,
- l'invariant de composant de base Inv_C qui a été déjà prouvé, et éventuellement un nouvel invariant Inv_CC_new exprimé au niveau du composite,
- la post-condition originale Post_origin qui est considérée comme garantie si la pré-condition originale est établie.

La clause **ASSERTIONS** contient la pré-condition originale Pre_origin et la post-condition promue Post_promoted.

6.5.2.2 Vérification

Pour chaque promotion d'un service offert promoted à partir d'un service origin , on construit une machine B comme indiqué dans le listing 6.7. Nous utilisons cette machine pour vérifier la correction de la promotion c'est-à-dire la correction du service au niveau du composite.

Proposition 6.5 (Contrat de promotion : CP) *La preuve des obligations liées aux assertions de la machine du listing 6.7 permet de démontrer l'obligation de preuve liée au contrat de promotion Kmelia que nous avons établi dans la définition 5.7 (**CP**).*

Preuve 6.5 *L'obligation de preuve associée aux assertions de la machine du* pattern *6.7 :*

$$\boxed{P \wedge I \Rightarrow J}$$

où P est le prédicat modélisant les propriétés, J une conjonction d'assertions, I l'invariant de la machine. Appliquée à la machine C_obs, l'obligation s'écrit alors :

$\equiv [\![Inv\_C]\!]_{\mathbb{B}} \wedge [\![Inv\_C\_new]\!]_{\mathbb{B}} \wedge [\![Post\_origin]\!]_{\mathbb{B}} \wedge [\![Pre\_promoted]\!]_{\mathbb{B}} \Rightarrow$
$[\![Pre\_origin]\!]_{\mathbb{B}} \wedge [\![Post\_promoted]\!]_{\mathbb{B}}$

$\equiv [\![Inv\_C]\!]_{\mathbb{B}} \wedge [\![Inv\_C\_new]\!]_{\mathbb{B}} \wedge [\![Pre\_promoted]\!]_{\mathbb{B}} \Rightarrow$
$([\![Pre\_origin]\!]_{\mathbb{B}} \wedge [\![Post\_promoted]\!]_{\mathbb{B}}) \Rightarrow [\![Post\_origin]\!]_{\mathbb{B}}$

$\equiv [\![Inv\_C]\!]_{\mathbb{B}} \wedge [\![Inv\_C\_new]\!]_{\mathbb{B}} \wedge [\![Pre\_promoted]\!]_{\mathbb{B}} \Rightarrow ([\![Pre\_origin]\!]_{\mathbb{B}} \Rightarrow$
$[\![Post\_origin]\!]_{\mathbb{B}}) \wedge [\![Post\_origin]\!]_{\mathbb{B}}$

Le contrat du service oigin est supposé correct, on obtient donc ;
$\equiv [\![Inv\_C]\!]_{\mathbb{B}} \wedge [\![Inv\_C\_new]\!]_{\mathbb{B}} \wedge [\![Pre\_promoted]\!]_{\mathbb{B}} \Rightarrow [\![Post\_origin]\!]_{\mathbb{B}}$

par rétroaction sur les éléments Kmelia on obtient :

$\equiv [\![[\![Inv\_C]\!]_{\mathbb{B}} \wedge [\![Inv\_C\_new]\!]_{\mathbb{B}} \wedge [\![Pre\_promoted]\!]_{\mathbb{B}}]\!]_{\mathbb{B}^{-1}} \Rightarrow [\![Post\_promoted]\!]_{\mathbb{B}}]\!]_{\mathbb{B}^{-1}}$

$\equiv Inv\_C \wedge Inv\_C\_new \wedge Pre\_promoted \Rightarrow Post\_promoted$

Ce qui établit la proposition 6.5. □

6.5.2.3 Des opérateurs pour assister la correction de la promotion

Ces opérateurs sont introduits à travers les mots-clés WEAKER, STRONGER, WITH précédant les prédicats des assertions du service origin afin de les renforcer ou les affaiblir dans la définition du service promoted.

1. **WEAKER** $predicate_{promoted}$
   - Introduit le prédicat $Predicate_{promoted}$ comme un affaiblissement du prédicat original.
   - L'obligation de preuve associée est :
   $$\boxed{predicate_{origin} \Rightarrow predicate_{promoted}}$$

2. **WEAKER WITH** $predicateW_{promoted}$
   - Le résultat de cette construction est un prédicat obtenu en appliquant la disjonction du prédicat original et un nouveau prédicat $predicateW_{promoted}$, c'est-à-dire $predicate_{promoted} \Leftrightarrow predicate_{origin} \vee predicateW_{promoted}$.
   - L'obligation de preuve associée :
   $$\boxed{predicate_{origin} \Rightarrow predicate_{promoted}}$$
   est vérifiée par construction. Notons que le prédicat $predicateW_{promoted}$ ne doit pas être déduit du prédicat original d'où l'obligation de preuve optionnelle :
   $\neg\ (predicate_{origin} \Rightarrow predicateW_{promoted})$.

3. **STRONGER** $predicate_{promoted}$
   - Introduit $predicate_{promoted}$ comme un renforcement du prédicat original.
   - L'obligation de preuve associée est :
   $$\boxed{predicate_{promoted} \Rightarrow predicate_{origin}}$$

4. **STRONGER WITH** $predicateW_{promoted}$
   - Construit $predicate_{promoted}$ sous forme d'une conjonction du prédicat original et un nouveau prédicat $predicateW_{promoted}$, c'est-à-dire $predicate_{promoted} \Leftrightarrow Predicate_{origin} \wedge predicateW_{promoted}$.
   - L'obligation de preuve associée :
   $$\boxed{predicate_{promoted} \Rightarrow predicate_{origin}}$$
   est vérifiée par construction. Du point de vue de la correction des assertions, le prédicat $predicate_{promoted}$ devrait être satisfaisable.

L'utilisation de ces opérateurs permet de capturer les intentions du concepteur, réduire l'effort de la preuve (lorsque le contrat est satisfait par construction) et dans certains cas, de détecter automatiquement les erreurs sans avoir recours à un assistant de preuve ; par exemple, il n'est pas nécessaire de vérifier l'affaiblissement d'une pré-condition originale d'un service offert ou le renforcement de sa post-condition si la post-condition originale ne dépend pas d'une partie de l'état ou des paramètres qui ont été contraints par le renforcement de la pré-condition.

Un autre opérateur additionnel **Weaker With** $predicate\_names_{remoted}$ peut être utilisé dans les modèles qui supportent des prédicats nommés. Cet opérateur permet de supprimer une partie de la conjonction des prédicats, ce qui permet un affaiblissement par construction.

Le tableau 6.5.4 montre l'impact des différents changements de prédicats lors de la promotion d'un service offert (ce tableau est une version plus complète du tableau 5.2.1).

Post \ Pré	Affaiblie	Renforcée		Inchangée
			With	
Post inchangée	✗	✔	✓	✓
Post affaiblie (With)	✗	✔	✓	✓
Post affaiblie	✗	✔	✔	✔
Post renforcée	✗	!	!	!

✓ : promotion sûre par construction,
✔ : promotion sûre,
✗ : promotion non sûre,
! : promotion souvent non sûre.

Tableau 6.5.4 – Modification des prédicats lors de la promotion d'un service offert

En nous basant sur les opérateur que nous avons introduit ci-dessus, nous avons caractérisé les obligations de preuve liées à la promotion en Kmelia :

1. **Cas de promotion non sûre :** le spécifieur reçoit immédiatement une notification indiquant que la promotion n'est pas correcte avant de lancer aucune preuve.
2. **Cas de promotion sûre par construction :** l'intention du spécifieur est capturée par les opérateurs de renforcement et d'affaiblissement et les obligations de preuve associées sont vérifiées par construction.
3. **Cas de promotion sûre :** comme dans le cas précédent, l'intention du spécifieur est capturée, mais pour que la promotion soit correcte, il faut prouver que les prédicats (pré/post-conditions) sont en conformité avec l'intention exprimée par le spécifieur. Autrement dit, prouver que la pré-condition (resp. post-condition) a été réellement renforcée (resp. affaiblie).
4. **Cas de promotion souvent non sûr :** dans ce cas, il faut prouver que les nouveaux prédicats satisfont le contrat de clientèle. Nous détaillons la preuve de ce cas dans la section 6.5.2.

6.6 Conclusion

Dans ce chapitre, nous avons montré comment vérifier la cohérence des composants, des assemblages et des composites Kmelia en se servant d'une plate-forme de preuve dédiée à la méthode B. La transposition de la vérification a été rendue possible par l'adoption d'une approche de vérification basée sur des contrats Pré/-Post d'une part, et d'une technique de *shallow embedding* [Att08] pour extraire des

spécifications B dont nous avons démontré l'équivalence en terme d'obligations de preuves générées d'autre part.

Nous extrayons alors systématiquement, sur la base de règles d'extraction que nous avons définies inductivement, des spécifications B à partir des composants ou assemblages Kmelia. Les machines B résultant de cette extraction sont fournies à l'AtelierB et soumises à la preuve.

L'originalité de cette approche est qu'elle intègre un processus de vérification modulaire où :

- on commence par prouver que les services satisfont leurs contrats ;
- on n'utilise ensuite que ces contrats, en abstrayant les services pour prouver que le composant les contenant est cohérent
- enfin, les composants, mis ensemble (assemblages ou composites) à travers ses services, satisfont bien les propriétés qu'on leur a associées.

Les expérimentations, que nous détaillerons dans le chapitre suivant, ont montré que les erreurs dans les spécifications Kmelia se traduisent par des obligations de preuve non déchargées. Elles peuvent alors être corrigées. Nous avons développé comme extension de la plate-forme COSTO, un module KML2B qui effectue cette extraction. Il sera également détaillé dans le chapitre suivant.

Chapitre 7

Instrumentation dans COSTO et expérimentations

L'objet de ce chapitre est de concrétiser l'ensemble des propositions autour de Kmelia faites dans cette thèse. Nous montrons donc comment sont implantées les différentes propositions présentées dans les chapitres précédents, tant pour l'intégration du langage de données et de contrats que pour l'extraction des spécifications B. L'ensemble de ces implantations a été intégré dans la plate-forme COSTO [1] dédiée au modèle Kmelia. L'exemple de gestion de stock vu dans le chapitre 4 est utilisé comme illustration.

7.1 Introduction

Au début de cette thèse, nous nous sommes fixés comme objectif le développement d'un prototype pour vérifier des propriétés de correction sur les composants logiciels et leurs assemblages. Ce prototype est indispensable pour expérimenter et valider nos techniques de vérification ainsi que la méthodologie globale évoquée dans le chapitre 5. En 2007, une version de la plate-forme COSTO telle qu'elle a été présentée dans le chapitre 3 était déjà développée [AAA07a]. À cette époque, COSTO comprenait essentiellement un analyseur statique des spécifications Kmelia ainsi que deux modules d'exportation vers MEC et LOTOS pour vérifier la partie dynamique, c'est-à-dire la compatibilité de l'interaction entre les services de différents composants. La figure 7.1 montre sa version actuelle.

Notons que lors des deux premières années de cette thèse, deux obstacles majeurs ont perturbé notre contribution à l'enrichissement de COSTO. D'une part, les travaux se sont multipliés autour de l'enrichissement du modèle Kmelia et donc de sa grammaire et, d'autre part, le noyau de COSTO était fortement dépendant de la grammaire qui a sans cesse continué à évoluer. Face à ces deux obstacles, il nous a paru plus intéressant de travailler sur une version plus ou moins stable de la grammaire Kmelia afin de commencer le développement du prototype. Cela explique également pourquoi certains éléments de la version actuelle de Kmelia ne sont pas encore pris en compte dans notre prototype.

1. COmponent Study TOolkit

Pour développer le prototype Kml2B, nous nous sommes basés sur l'environnement de développement intégré Eclipse. Eclipse est un IDE (*Integrated development environment*) libre, multi-langage et extensible. La spécificité d'Eclipse provient de son découpage en *plugins*. Chaque fonctionnalité d'Eclipse est ainsi isolée dans des plugins séparés. De plus, Eclipse propose un ensemble de librairies permettant à des développeurs tiers de créer leurs propres extensions. Cela nous a largement facilité la collaboration avec les autres acteurs impliqués dans le développement de COSTO, en particulier Gilles Ardourel, chargé du noyau COSTO.

7.2 La nouvelle plate-forme COSTO

Les analyses de propriétés des spécifications Kmelia sont mises en œuvre dans la plateforme d'expérimentation COSTO [AAA07a], sous forme d'un ensemble de *plugins* Eclipse. On distingue principalement deux sortes de plugins, à savoir les plugins d'environnement et les plugins d'exportation. La figure 7.1 donne une représentation simplifiée de l'architecture de COSTO.

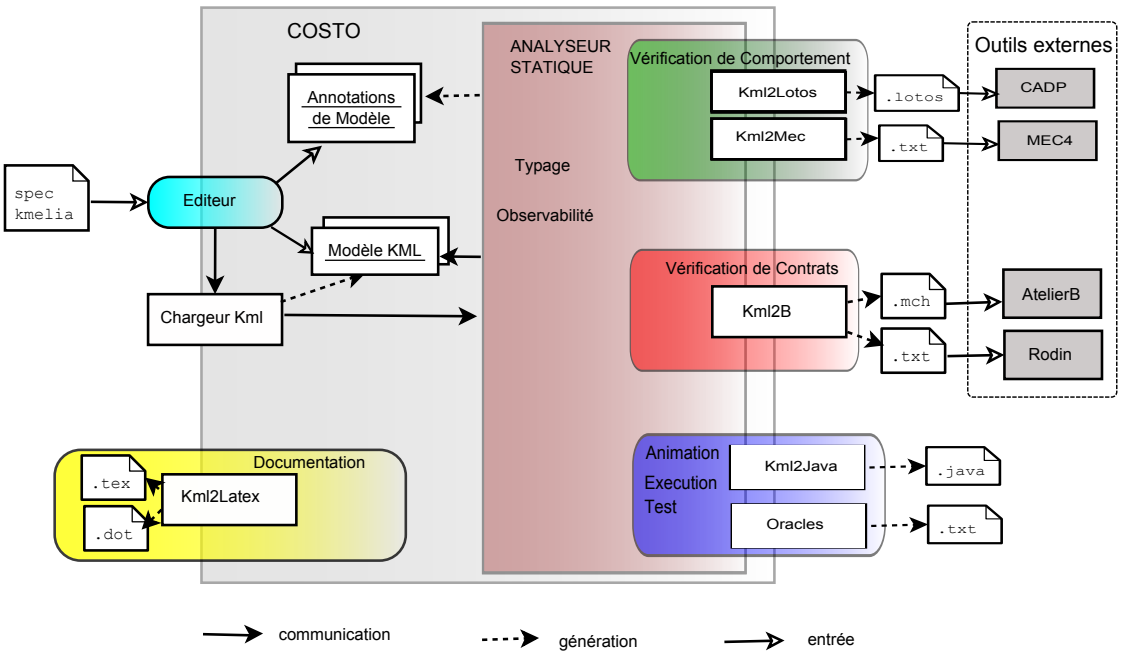


FIGURE 7.1 – Aperçu de l'architecture de COSTO

Dans la suite de cette section, nous présentons les différents plugins intégrés dans la plate-forme COSTO.

7.2.1 Plugins d'environnement

Le noyau

Ce plugin représente le cœur de la plate-forme COSTO, il comprend essentiellement la grammaire antlr[1] de Kmelia et un modèle d'objet interne (ensemble de classes Java) permettant de construire un arbre abstrait à partir d'une spécification Kmelia. Cet arbre est généré par le chargeur de spécifications Kmelia (*loader*) et est utilisé comme entrée pour tous les autres plugins.

Le noyau intègre aussi un analyseur qui assure non seulement des vérifications classiques (syntaxe et contrôle de type) mais également des vérifications spécifiques à Kmelia comme la cohérence des interfaces, l'observabilité, la formation correcte des systèmes de transition.

L'interface graphique

Il s'agit d'un plugin développé principalement pour faciliter la manipulation des spécifications Kmelia. Il propose un éditeur basé sur Eclipse donnant accès à de nombreuses fonctionnalités telles que la complétion automatique, la coloration syntaxique des mots-clés Kmelia, le marquage des erreurs, *etc.*

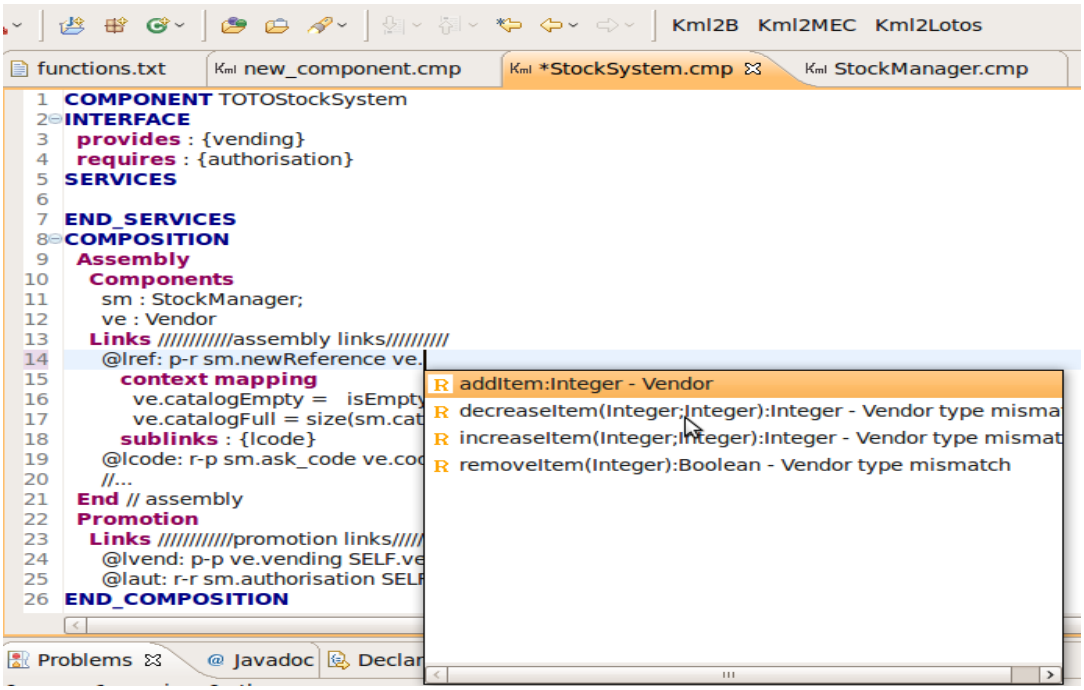


FIGURE 7.2 – Menu contextuel activé pour la complétion automatique des services d'un composant

La figure 7.2 montre un exemple de complétion lors de la définition d'un lien d'assemblage. Dans cet exemple, seuls les services requis définis dans le composant ve (instance du composant type Vendor) sont proposés dans un menu déroulant. De

1. http ://www.antlr.org/

plus, certaines informations relatives à l'analyse statique apparaissent d'ores et déjà sur le menu déroulant. Par exemple, le message "*Vendor type mismatch*" avertit le spécifieur que seul le service requis addItem est compatible (vis-à-vis de la signature) avec le service offert newReference.

De même, la figure 7.3 montre un exemple de marquage des erreurs. Dans cet exemple, trois erreurs de natures différentes sont signalées : les deux premières sont liées à la vérification spécifique à Kmelia (par exemple le non-respect des règles d'observabilité présentées dans la section 4.4.1). La troisième est une erreur typiquement syntaxique.

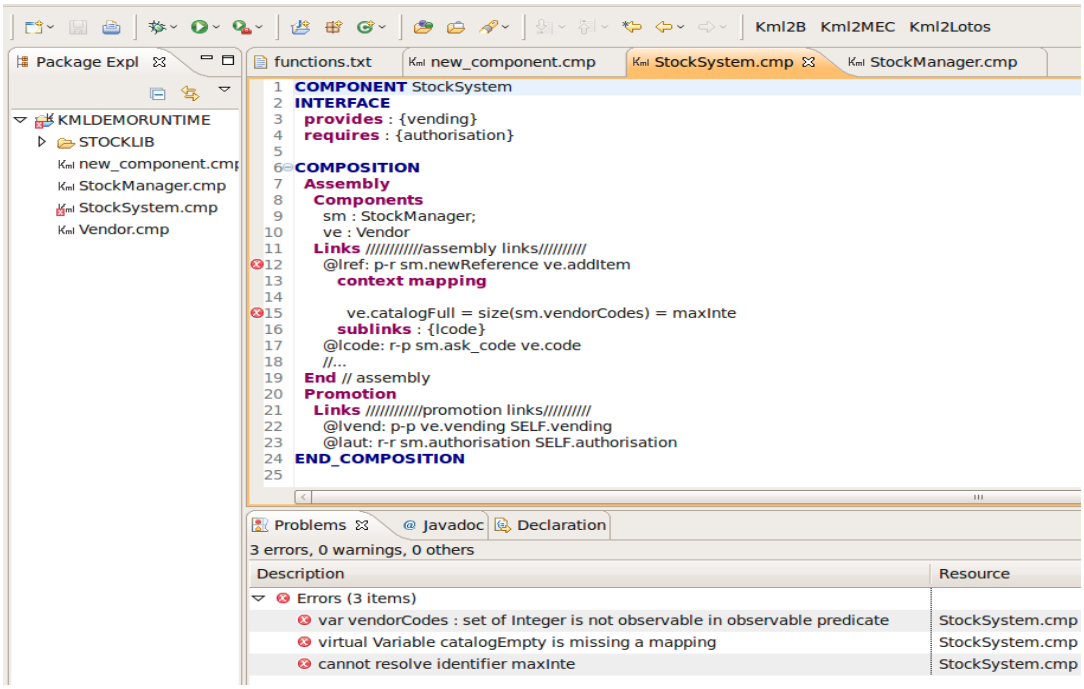


FIGURE 7.3 – Exemples d'erreurs détectées par l'analyseur de COSTO

7.2.2 Plugins d'exportation

Kml2Lotos et **Kml2MEC**. Comme nous l'avons indiqué plus haut, ces deux plugins ont déjà été développés dans la version initiale de la plate-forme COSTO. Le but est de traduire le comportement des services Kmelia en MEC et/ou LOTOS pour la vérification des interactions entre les systèmes de transition qui les modélisent. Le principe est de s'appuyer sur des environnements existants afin d'exploiter au mieux les résultats connus en terme d'analyse de la dynamique des systèmes, et plus généralement, de vérification (par exemple, le *model checker* de l'outil CADP/Lotos).

Kml2B. Il s'agit d'un plugin que nous avons développé dans le cadre de cette thèse. Il permet d'extraire des spécifications B à partir des composants (ou composites) et des assemblages Kmelia. Il joue le rôle d'une passerelle entre COSTO et les outils associés à la méthode B. Les spécifications Kmelia peuvent

donc être analysées avec ces derniers de façon transparente. Ce plugin sera détaillé dans la section 7.3.

Kml2Jml. Il s'agit d'un plugin que nous sommes en train de développer afin d'expérimenter la solution que nous proposons pour vérifier la correction fonctionnelle des services. Comme nous l'avons signalé dans la section 6.2, nous ciblons l'outil KeY [1] (associé au langage JML [2]) basé sur l'exécution symbolique. Le but de ce plugin est alors de générer des spécifications JML à partir des services Kmelia.

Kml2Java. Ce plugin permet la traduction des composants Kmelia en des classes Java permettant par la suite l'animation de spécifications Kmelia. Le résultat sera également exploité pour tester les composants Kmelia.

Kml2LaTeX. Ce plugin permet de produire la documentation en LaTeX des spécifications Kmelia. Il comprend la description des composants et de leurs assemblages. Il génère également un graphe (.dot et .eps) des automates de services.

La figure 7.4 montre une vue globale de l'interaction entre la pate-forme Kmelia/COSTO et les outils externes.

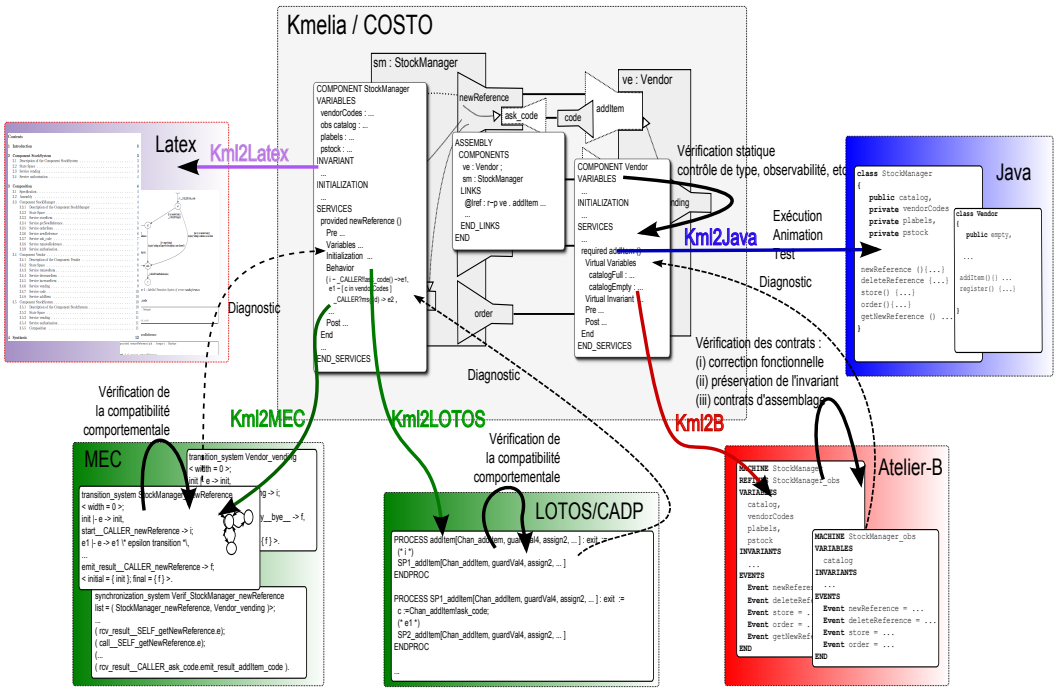


FIGURE 7.4 – Aperçu du *framework* Kmelia/COSTO

7.3 Le plugin Kml2B

Le but de ce plugin est d'extraire des machines B à partir des spécifications Kmelia. D'abord on génère un arbre syntaxique abstrait AST(*Abstract Syntax Tree*)

1. http ://key-project.org/
2. Java Modeling Language

en se basant sur le *loader* Kmelia. Chaque nœud de cet arbre est représenté par une classe dans la structure d'objet interne. L'arbre abstrait fait l'objet d'une exploration afin de traduire chaque nœud en un autre équivalent dans l'AST cible (B en l'occurrence).

7.3.1 Implantation basée sur le patron *visiteur*

Afin d'implanter les algorithmes de traduction en B présentés dans le chapitre 6, nous avons utilisé le patron *visiteur* [GHJV95]. Un visiteur représente une opération à effectuer sur les éléments d'une structure d'objets. Le choix de ce patron est motivé par les besoins suivants :

- la structure d'objets formant le noyau de COSTO contient de nombreuses classes, et l'on veut réaliser des opérations sur ces dernières. Le visiteur permet de définir ces opérations sans modifier les classes des éléments sur lesquels il opère ;
- nous souhaitons que le plugin soit suffisamment indépendant de la structure d'objet interne, de manière à ce que le code métier ne soit pas pollué par les opérations liées à l'objet du visiteur (extraction dans le cas de Kml2B) ;
- le visiteur permet de regrouper toutes les opérations du même type dans une seule classe, il est donc plus facile d'envisager d'autres opérations (par exemple l'extraction vers Event-B ou vers JML ou toute autre opération) sur la structure d'objet interne.

7.3.2 Principes

Il s'agit de définir deux opérations applicables à la hiérarchie de classes modélisant les différents nœuds de l'AST Kmelia. Pour ce faire, il a fallu distinguer la traduction des expressions et la traduction des éléments de modélisation. Il suffit simplement de créer deux visiteurs concrets correspondant chacun à une fonctionnalité donnée et de définir un visiteur abstrait (cf. figure 7.5). Chaque visiteur concret contient autant de méthodes, correspondant à l'opération qu'il définit, que de classes concrètes représentant l'AST considéré. Ces méthodes seront ensuite invoquées et appliquées sur les classes de l'AST uniquement par l'intermédiaire de l'opération accept() qui doit être ajoutée à chacune de ces dernières. Les méthodes accept() prennent en paramètres les visiteurs concrets, auxquels elles délèguent les requêtes en invoquant les méthodes appropriées aux classes de l'AST.

Dans la figure 7.5 :

- Kml2BVisitor déclare une opération visitKmlElementX() pour chaque classe d'un élément concret KmlElementX ou KmlExpression. Le nom de l'opération et sa signature identifient la classe émettrice de la requête vers le visiteur, qui peut ensuite accéder directement à l'élément, à travers son interface particulière. Notons que les expressions Kmelia (KmlExpression) sont mises en évidence car elles font l'objet d'une exploration spécifique par le visiteur KmlExpression2B ;
- KmlElement2BVisitor et KmlExpression2BVisitor concrétisent par codage chaque opération déclarée par la classe Kml2BVisitor ;

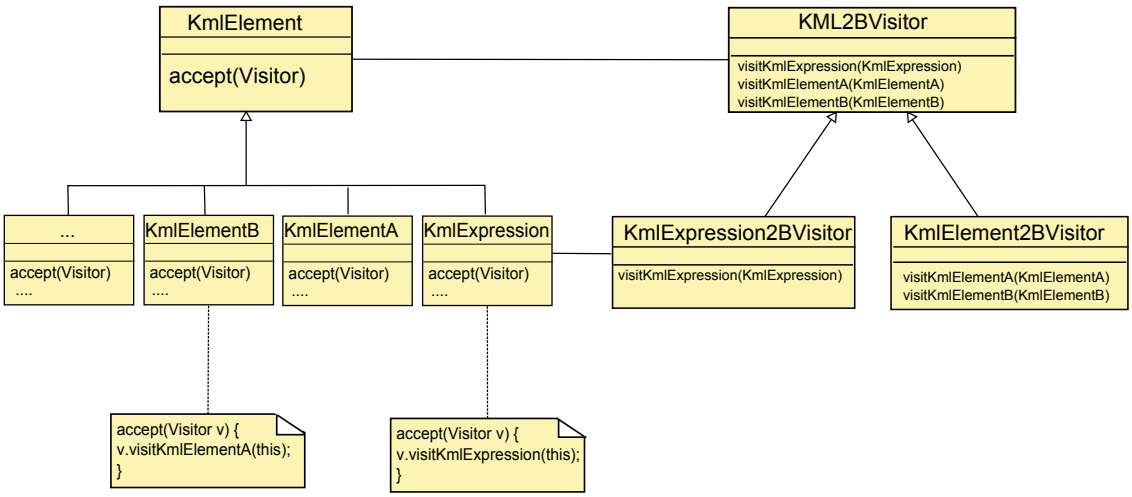


FIGURE 7.5 – Extrait de la structure d'implantation de Kml2B basée sur le patron *visiteur*.

- KmlElement définit une opération accept() qui prend pour argument un visiteur ;
- KmlElementX et KmlExpression réalisent le codage de l'opération accept().

Ainsi, les opérations liées à la traduction des expressions Kmelia en B (ou à l'extraction des spécifications B) sont isolées et regroupées dans une seule classe. Le code correspondant est donc moins dispersé (à l'exception de l'opération accept()) et plus lisible. Il est plus facile ainsi de maintenir, de faire évoluer et de réutiliser ces fonctionnalités. Le patron *visiteur* permet donc de rajouter facilement de nouvelles opérations sur la hiérarchie de l'AST, en ajoutant tout simplement un nouveau visiteur concret.

7.3.3 Utilisation de Kml2B

Afin d'utiliser le plugin Kml2B, il faut d'abord télécharger les plugins **Antlr**, **Costo_Core**, **Costo_UI** et **Costo_Kml2B** et les placer dans le répertoire `/dropins/plugins` d'Eclipse, ensuite lancer Eclipse, et puis, dans l'onglet à gauche, naviguer jusqu'au répertoire du projet contenant les spécifications Kmelia à analyser.

À ce stade, trois contextes d'utilisation du plugin Kml2B peuvent être distingués (cf. figure 7.6). Le premier est considéré quand la spécification Kmelia décrit un composant primitif, les deux autres concernent la composition des composants par des liens d'assemblage ou de promotion. Notons que la partie liée à l'extraction des machines à partir de la promotion de services est encore en cours de développement et donc n'est pas détaillée ici. Quel que soit le contexte, une compilation est lancée automatiquement après le chargement de la spécification et les erreurs, s'il y en a, apparaîtront sur le bord de l'éditeur.

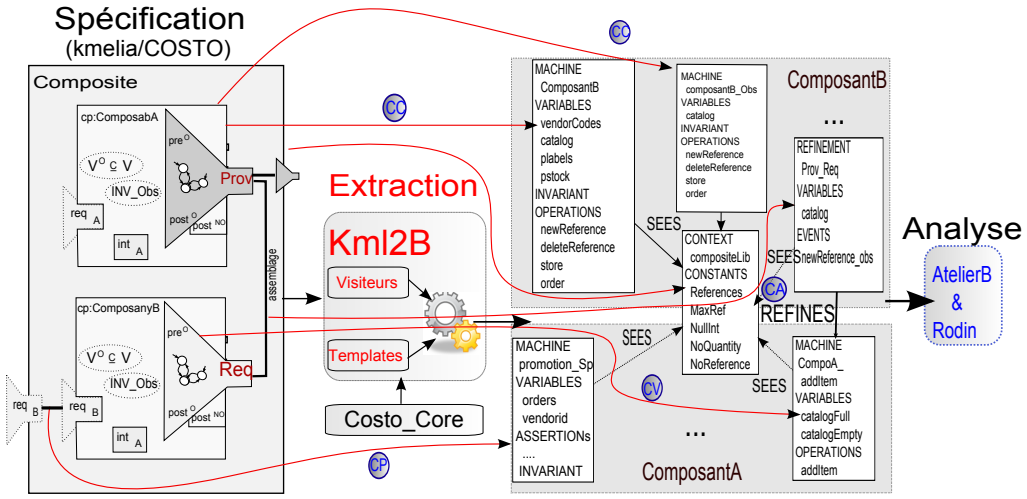


FIGURE 7.6 – Aperçu du plugin Kml2B.

Vérification de la cohérence d'un composant. Nous rappelons que quelle que soit l'architecture d'une application, sa spécification Kmelia est toujours une description d'un composant. Ce dernier est soit primitif, soit un composite contenant un ou plusieurs assemblages. Dans un premier temps, en cliquant sur le bouton Kml2B, deux machines B sont systématiquement générées : la première pour vérifier la cohérence de la partie observable de ce composant (cf. la propriété **CO** de la proposition 6.1) et la seconde pour vérifier sa cohérence vis-à-vis de ses services (cf. la propriété **CC** de la proposition 6.2).

Vérification de la compatibilité des assertions dans un assemblage. Si le composant contient, au moins, un assemblage, Kml2B affiche la liste des liens d'assemblage définis dans la spécification, puis demande à l'utilisateur d'en sélectionner un, sur lequel il souhaite vérifier le contrat d'assemblage.

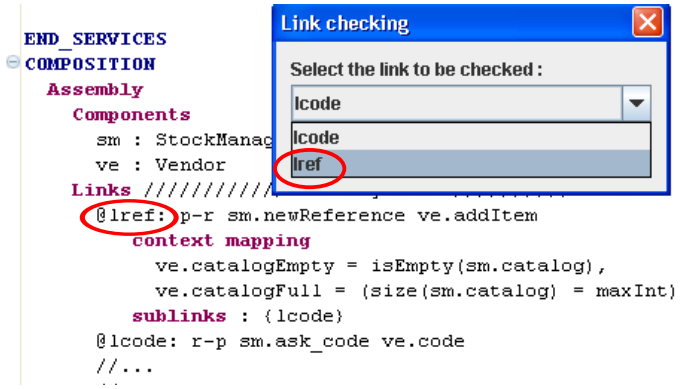


FIGURE 7.7 – Choix d'un lien d'assemblage pour l'extraction de machines B

Par exemple, dans la figure 7.7, on voit la liste des liens d'assemblage, affichée lors de la traduction du composant StockSystem. L'utilisateur peut choisir le lien

étiqueté Iref pour vérifier le contrat d'assemblage entre le service requis addItem du composant Vendor et le service offert newReference du composant StockManager. Kml2B génère, pour le lien sélectionné, une machine pour le service requis et un raffinement pour le service offert. La première est utilisée pour vérifier la cohérence du contexte virtuel du service requis (cf. la propriété **CV** de la proposition 6.3). Le raffinement sert à vérifier le contrat d'assemblage (cf. la propriété **CA** de la proposition 6.4).

7.4 Retour sur l'exemple du *Stock Management*

Afin d'illustrer les mécanismes liés à l'extraction de spécification B et à la procédure de vérification, nous revenons sur l'exemple de gestion de stock présenté dans la section 4.3. La figure 7.8 rappelle l'architecture Kmelia simplifiée de cet exemple.

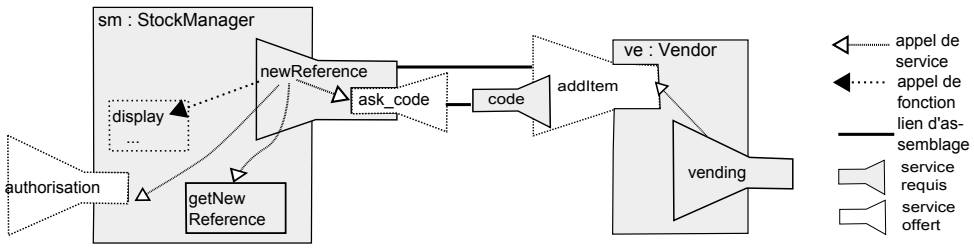


FIGURE 7.8 – Architecture Kmelia simplifiée de l'application de gestion de stock

7.4.1 Vérification de l'invariant du composant

Comme cela est indiqué dans la section 6.2, plusieurs machines B sont générées pour vérifier les différents niveaux de cohérence du composant.

Listing 7.1 – La machine B modélisant la bibliothèque StockLib

```
MACHINE
  StockLib

SEES
  Default

CONSTANTS
  MaxRef,
  References,
  NoQuantity,
  NoReference

PROPERTIES
  MaxRef = 100 ∧
  References = 1..MaxRef ∧
  NoQuantity = −2 ∧
  NoReference = −3

END
```

Étape 1. Génération d'une machine modélisant la bibliothèque de données partagée comme StockLib (cf. listing 7.1) qui sera vue par toutes les autres machines.

Dans la suite, nous nous focalisons sur la spécification du composant StockManager (cf. listing dans la section 4.3), qui définit un ensemble de variables, en particulier la variable *observable* catalog modélisant l'ensemble des références des articles. Deux tableaux (plabels et pstock) sont utilisés pour stocker les étiquettes des références actuellement disponibles et leur quantité de stock. L'invariant du composant stipule que le catalog a une taille maximale ; chaque référence dans le catalog a une étiquette et une quantité ; une référence inconnue ne doit avoir d'entrée dans aucun des deux tableaux pstock et plabels . La vérification de la cohérence du composant StockManager se fait en appliquant la méthode présentée dans la section 6.3.3.

Étape 2. Une machine B StockManager_obs est d'abord générée suivant le *pattern* du listing 6.3 : seule la variable observable catalog du composant devient une variable dans la machine. La partie observable de l'invariant est aussi traduite en suivant les règles présentées dans la section 6.3.2. Enfin, la machine contient une opération pour chaque service offert. La machine StockManager_obs est utilisée pour prouver la cohérence du composant vis-à-vis de ses services en se limitant aux parties observables (propriété **CO** de la proposition 6.1).

Listing 7.2 – La machine B StockManager_obs générée

```
MACHINE
  StockManager_obs

SEES
    Default , StockLib

VARIABLES
  catalog /* obs */

INVARIANT
  catalog ∈ 𝔽(References) ∧ /*obs*/
  /*@borned*/
  card(catalog) ≤ MaxRef /*obs*/

INITIALISATION
  catalog := ∅

OPERATIONS
  Result ⟵ newReference =
  PRE
    Result ∈ ℤ ∧
    card(catalog) < MaxRef /*obs*/
  THEN
    ANY l_Result, l_catalog
    WHERE
      l_Result ∈ ℤ ∧
      l_catalog ∈ 𝔽(References)
      ∧
      ( (l_Result > 0 ∧ l_Result ≤ MaxRef)
                or l_Result = NoReference )/*obs*/
      ∧
      ( l_Result ≠ NoReference ⇒
        l_Result /∈ catalog ∧
        l_catalog = catalog ∪ {l_Result})/*obs*/
      ∧
      ( l_Result = NoReference ⇒
        l_catalog = catalog ) /*obs*/
    THEN
      Result := l_Result ||
      catalog := l_catalog
    END
  END

END
```

Étape 3. Une autre machine B StockManager est construite de la même façon que StockManager_obs mais en suivant le *pattern* du listing 6.4. Cette machine prend en compte la partie interne (non-observable) de l'espace d'état du composant en incluant tous les éléments non-observables (variables et invariant, *etc.*). Les opérations correspondant aux services offerts ajoutent les post-conditions locales à leur définition. La machine StockManager est utilisée pour vérifier les propriétés **CC** et **CC'** (proposition 6.2).

Listing 7.3 – La machine B StockManager

```
MACHINE
  StockManager

SEES
    Default , StockLib

VARIABLES
  vendorCodes,
  catalog , /* obs */
  plabels ,
  pstock ,
  Result_newreference

INVARIANT
  vendorCodes ⊆ ℤ ∧
  catalog ∈ 𝔽(References) ∧ /*obs*/
  plabels ∈ References → String ∧
  pstock ∈ References → ℤ ∧
  Result_newreference ∈ INT ∧
  /*@borned*/  card(catalog) ≤ MaxRef ∧        /*obs*/
  /*@referenced*/
  ∀ ref1 . (ref1 ∈ References ∧ ref1 ∈ catalog ⇒ (plabels(ref1) ≠ EmptyString
                                                  ∧ pstock(ref1) ≠ NoQuantity))
  ∧   /*@notreferenced*/
  ∀ ref2 . ((ref2 ∈ References ∧ ref2 /∈ catalog) ⇒ ((plabels(ref2) = EmptyString
                                                  ∧ pstock(ref2) = NoQuantity)))

INITIALISATION
  vendorCodes := ∅ ||
  catalog := ∅ ||
  plabels := (1..MaxRef) × {EmptyString} ||
  pstock := (1..MaxRef) × {NoQuantity}||
  Result_newreference := 0

OPERATIONS
  Result ⟵ newReference =
    PRE
        card(catalog) < MaxRef
    BEGIN
    ANY new_Result, new_catalog, new_pstock, new_plabels
    WHERE
      new_Result ∈ ℤ ∧
      new_catalog ∈ 𝔽(References) ∧
      new_plabels ∈ References → String ∧
      new_pstock ∈ References → ℤ ∧
      new_Result ≠ NoReference ∧
      ( (new_Result > 0 ∧ new_Result ≤ MaxRef) ∨ new_Result = NoReference ) /*obs*/
      ∧       ( new_Result ≠ NoReference ⇒
          (  new_catalog = catalog ∪ {new_Result} ) ) /*obs*/
      ∧ ( new_Result = NoReference ⇒  new_catalog = catalog ) /*obs*/
      ∧       ( new_Result ≠ NoReference ⇒
          (new_pstock(new_Result) = 0 ∧
          new_plabels(new_Result) ∈ String−{EmptyString} ∧
          ∀ ii . (ii ∈ 1..MaxRef ∧ ii ≠ new_Result ⇒
              (new_pstock(ii) = pstock(ii) ∧ new_plabels(ii) = plabels(ii) ))))
      ∧ ( new_Result = NoReference ⇒ ( ∀ ii . (ii ∈ 1..MaxRef ⇒
            (new_pstock(ii) = pstock(ii) ∧ new_plabels(ii) = plabels(ii) )) ))/*obs*/
    THEN
        Result := new_Result ||
        Result_newreference := new_Result ||
        catalog := new_catalog ||
        pstock := new_pstock ||
        plabels := new_plabels
    END
  END
END
```

Étape 4. Pour chaque service requis du composant StockManager, une machine B spécifique est générée selon le *pattern* du listing 6.5 afin de vérifier la cohérence du contexte virtuel du service en question. Cette machine est utilisée pour vérifier la propriété **CV** (propriété de la proposition 6.3).

La cohérence du composant Vendor et de ses services offerts/requis est vérifiée en appliquant le même processus. Nous nous intéressons plus particulièrement au service requis addItem et à la machine correspondante Vendor_addItem obtenue en appliquant le pattern 6.5.

Listing 7.4 – La machine addItem

```
MACHINE
  addItem_mch
SEES
    StockLib

VARIABLES
  catalogFull ,
  catalogEmpty,
  Result_addItem

INVARIANT
  catalogFull  ∈ BOOL ∧
  catalogEmpty ∈ BOOL ∧
  Result_addItem ∈ ℤ ∧
  ¬ (catalogEmpty = TRUE ∧ catalogFull = TRUE)

INITIALISATION
  ANY
    catFull , catEmpty
  WHERE
    catFull  ∈ BOOL ∧
    catEmpty ∈ BOOL ∧
    ¬ (catEmpty = TRUE ∧ catFull = TRUE)
  THEN
    catalogFull  := catFull ||
    catalogEmpty := catEmpty ||
    Result_addItem := 0
  END

OPERATIONS
  result ⟵ addItem =
BEGIN
        ANY
          new_result, new_catalogEmpty, new_catalogFull
        WHERE
          new_result ∈ ℤ ∧
          new_catalogEmpty ∈ BOOL ∧
          new_catalogFull ∈ BOOL
          ∧
          ( new_result ≠ NoReference ⇒
            new_catalogEmpty = FALSE ∧ new_catalogFull ∈ BOOL )
          ∧
          ( new_result = NoReference ⇒
            new_catalogEmpty = catalogEmpty ∧ new_catalogFull = catalogFull )
        THEN
            result  := new_result ||
            Result_addItem := new_result ||
          catalogEmpty := new_catalogEmpty ||
           catalogFull  := new_catalogFull
        END
  END

END
```

7.4.2 Vérification des liens d'assemblage

Nous avons utilisé la méthode de vérification présentée dans la section 6.5.1.2 pour vérifier le lien d'assemblage entre le service requis addItem du composant Vendor et le service offert newReference du composant StockManager.

Étape 5. Un raffinement B addItem_newReference_ref est généré pour vérifier la correction du contrat d'assemblage. Ce dernier raffine la machine Vendor_addItem précédemment générée. Son invariant inclut l'invariant observable du composant StockManager et le *mapping* entre les variables virtuelles du service addItem et la variable observable catalog du composant StockManager.

Listing 7.5 – Le raffinement new_reference_ref

```
REFINEMENT
   new_reference_ref

REFINES
   addItem_mch

SEES
   StockLib

VARIABLES
    catalogEmpty,
    catalogFull ,
    Result_addItem ,
    catalog

INVARIANT
    catalog ∈ ℙ(References) ∧
    card(catalog) ≤ MaxRef ∧
    catalogEmpty = bool( card(catalog) = 0 ) ∧
    catalogFull = bool( card(catalog) = MaxRef ) ∧
    (¬ ( catalogFull = TRUE )⇒card(catalog) < MaxRef)∧
    Result_addItem ∈ ℤ

INITIALISATION
    catalogFull := FALSE ||
    catalogEmpty := TRUE ||
    Result_addItem :=0 ||
    catalog := ∅

OPERATIONS
   result ⟵ addItem =
   PRE
      card(catalog) < MaxRef  /* après correction */
   THEN
      ANY new_result , new_catalog, new_catalogEmpty , new_catalogFull
      WHERE
          new_result ∈ ℤ ∧ new_catalogEmpty ∈ BOOL ∧ new_catalogFull ∈ BOOL ∧
            new_catalog ∈ ℙ(References) ∧ (new_catalogEmpty = bool(card(new_catalog)=0))
            ∧ (new_catalogFull = bool(card(new_catalog)=MaxRef))
          ∧ (((new_result>0 ∧ new_result ≤ MaxRef) ∨ new_result = NoReference)
                      ∧ (new_result= NoReference ⇒ new_catalog = catalog))
          THEN
                       result := new_result ||
                      Result_addItem := new_result||
          catalogEmpty := new_catalogEmpty ||
           catalogFull := new_catalogFull
      END
   END
END
```

L'analyse de cet exemple avec l'AtelierB a révélé un certain nombre d'erreurs. Par exemple, le fait que la pré-condition réduite à $True$ dans le service requis addItem a violé le contrat d'assemblage

$$\boxed{\text{Pre(addItem)} \Rightarrow \text{Pre(newReference)}}$$

car (True $\Rightarrow \neg$ (size(catalog) $<$ MaxRef)) n'est pas toujours vrai. La figure 7.9 montre l'obligation de preuve non prouvée par l'atelierB due à la violation du contrat d'assemblage.

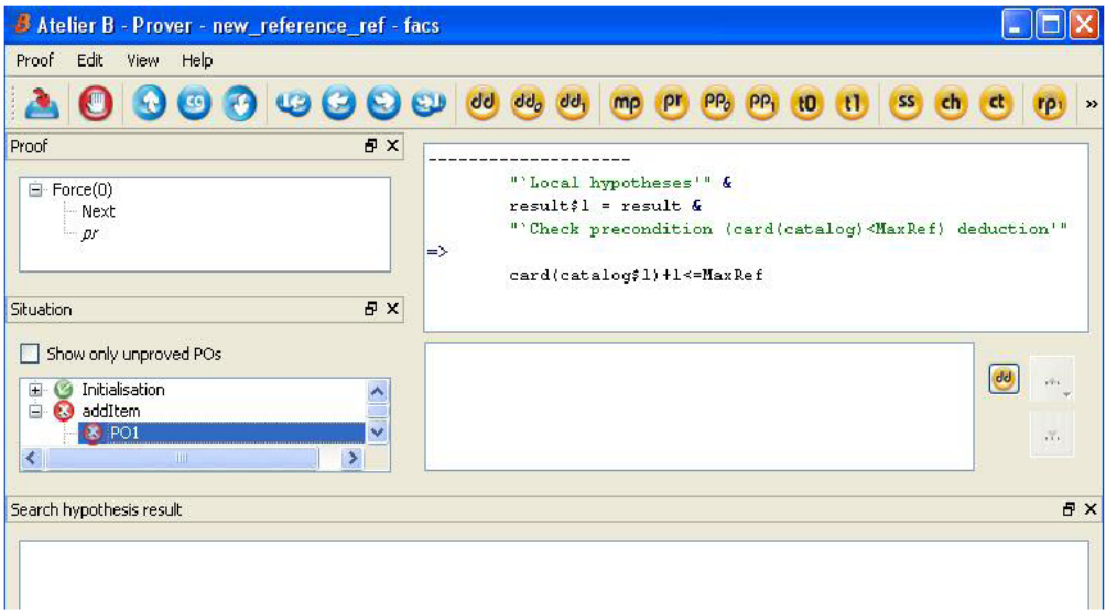


FIGURE 7.9 – Trace d'un échec de preuve avec l'AtelierB du contrat d'assemblage entre le service addItem et le service newReference.

Après interprétation de cet échec, deux solutions sont envisageables : soit affaiblir la pré-condition du service newReference, soit renforcer celle du service addItem.

Dans un premier temps, nous avons choisi la première solution : la pré-condition du service newReference a été affaiblie à $True$. Cela nous a permis de prouver la correction du lien d'assemblage mais pas l'assemblage, car le composant StockManager n'est désormais plus cohérent. Cela est dû au fait que son invariant size (catalog) $\leq$ MaxRef n'est plus préservé par la post-condition (voir la figure 7.10).

La deuxième solution a consisté à renforcer la pré-condition du service addItem par le prédicat $\neg$ catalogFull. Ainsi, comme le montre la figure 7.11, nous avons prouvé l'assemblage tout en préservant la cohérence du composant StockManager.

7.4.3 Vérification des liens de promotion

D'un point de vue méthodologique, le service d'un composant ne devrait pas appeler directement un service d'un sous-composant, mais plutôt le service issu de sa promotion. Cependant le spécifieur n'est pas obligé de mettre chaque service

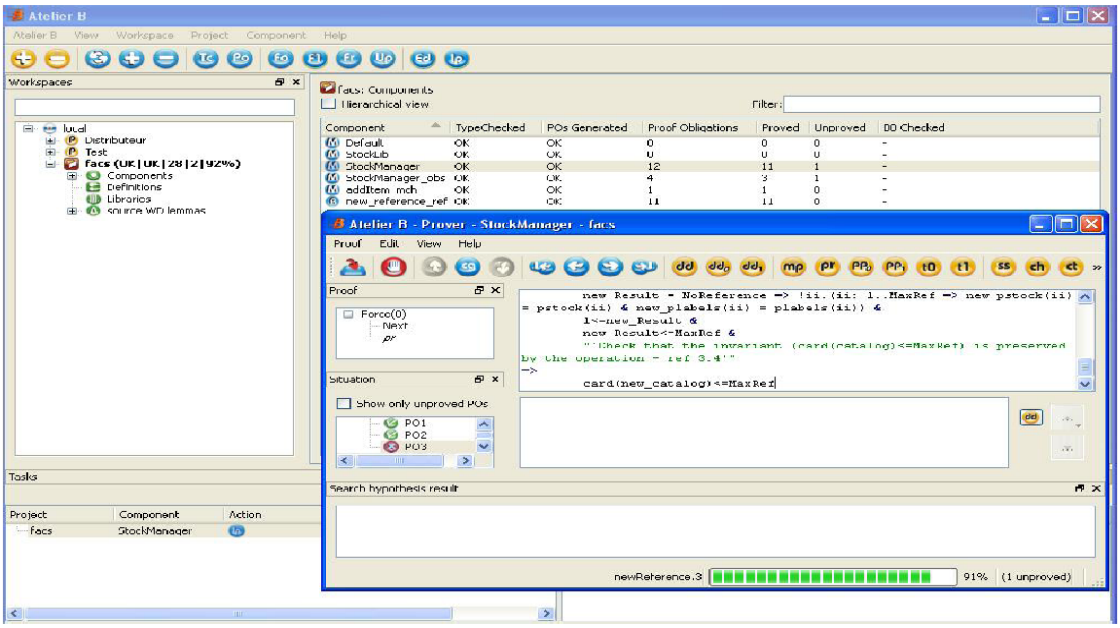


FIGURE 7.10 – Trace d'un échec de preuve avec l'AtelierB de la cohérence du composant StockManager.

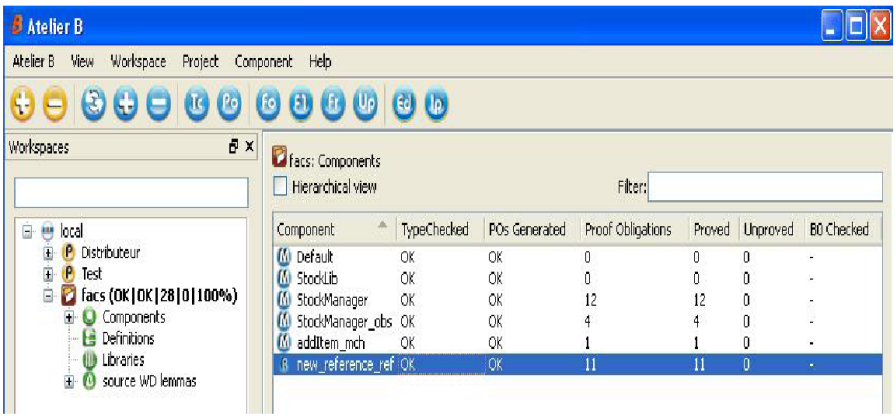


FIGURE 7.11 – Synthèse des obligations de preuve générées/prouvées avec l'AtelierB

promu dans l'interface du composite sauf si besoin est. La figure 7.12 met en évidence l'architecture interne du composite *StockManager*.

Étape 6. La machine addToEntry_Store du listing 7.6 est utilisée pour vérifier la correction du service Store dans le composite StockManager, qui est le résultat de la promotion du service addToEntry du composant IntDictionary . Le contenu de la machine est une instance du *pattern* du listing 6.7.

Six obligations de preuve sont générées à l'issue de l'analyse de cette machine. En utilisant l'AtelierB, nous avons prouvé la totalité des obligations de preuve en mode *Automatic force (1)*.

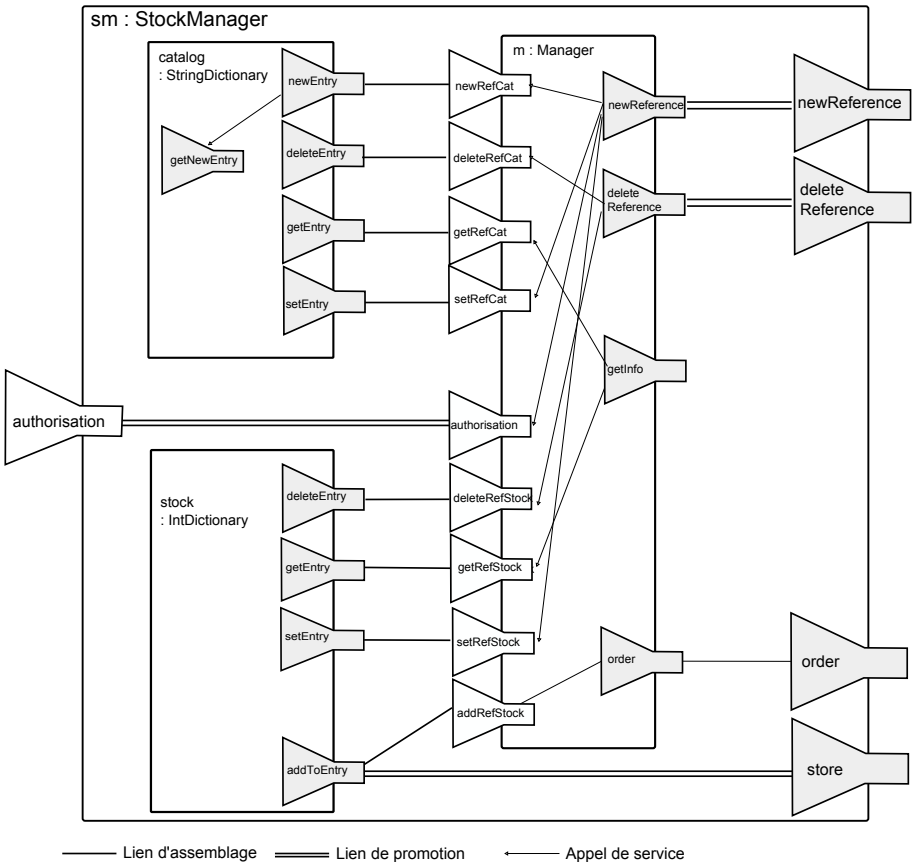


FIGURE 7.12 – Description du composite Stock Manager

Cependant, pour mieux illustrer notre approche, nous avons volontairement introduit des erreurs dans les contrats des services promus, à savoir :

- la pré-condition du service promu a été affaiblie en supprimant le prédicat pid >= 10 de la pré-condition originale. Dans ce cas, quelle que soit la post-condition, il restait toujours des obligations de preuve non prouvées par l'AtelierB. Cela correspond à la colonne "non sûre" dans le tableau 5.2.1.
- garder la pré-condition inchangée et ne pas introduire d'autres restrictions dans l'invariant du composite StockManager. Dans ce cas, l'AtelierB n'a pas pu prouver la post-condition renforcée du service promu car il n'a aucune garantie que pqty > 0 (pré-condition) établisse le prédicat pstock[pid] < old(pstock)[pid]. Cela correspond au premier cas de "souvent non sûre" dans le tableau 5.2.1. Ce cas simple pourrait être détecté même sans lancer le prouveur (cf. tableau 6.5.4).
- renforcer la pré-condition par pqty > 0 et la post-condition par pstock[pid] < old(pstock)[pid]. Cela rend la nouvelle post-condition non satisfaisable. Par conséquent, l'AtelierB ne peut pas la prouver.

La figure 7.13 donne une vue globale des machines B devant être générées afin de vérifier les spécifications Kmelia.

Listing 7.6 – Machine StockManager_addToEntry_store générée pour vérifier la promotion du service addToEntry

```
MACHINE
    StockManager_addToEntry_store
CONCRETE_CONSTANTS
    /* origin variables */
        o_keys ,
        o_values ,
        o_result ,
        o_noValue ,
    /* promoted variables */
        p_catalog ,
        p_labels ,
        p_stock ,
        p_result ,
    /* origin parameters */
        o_key ,
        o_value ,
    /* promoted parameters */
        p_id ,
        p_qty ,
    /* updated variables*/
        l_o_values ,
        l_p_stock
PROPERTIES
    /* origin typing */
        o_keys ⊆ 1 .. 100 ∧
        o_values ∈ 1 .. 100 → INT ∧
        o_result ∈ INT ∧
        o_noValue ∈ INT ∧
        o_noValue = − 1 ∧
    /* promoted typing */
        p_catalog ⊆ 1 .. 100 ∧
        p_labels ∈ 1 .. 100 → INT ∧
        p_stock ∈ 1 .. 100 → INT ∧
        p_result ∈ INT ∧
    /* origin parameters typing */
        o_key ∈ INT ∧
        o_value ∈ INT ∧
    /* promoted parameters typing */
        p_id ∈ INT ∧
        p_qty ∈ INT ∧
    /* updates variables typing */
        l_o_values ∈ 1 .. 100 → INT ∧
        l_p_stock ∈ 1 .. 100 → INT ∧
    /* origin invariant already proved */
        card ( o_keys ) ≤ 100 ∧
    /* origin post */
        o_result = o_key ∧
        l_o_values(o_key) = o_values(o_key)+
                                o_value ∧
    /* maping predicat */
        p_result = o_result ∧
        p_catalog = o_keys ∧
        p_stock = o_values ∧
        p_id = o_key ∧
        o_value = p_qty ∧
        l_o_values = l_p_stock ∧
    /* pre promoted */
        ( p_id ∈ p_catalog ∧ p_stock ( p_id ) +
                    p_qty ≥ 0 ∧ p_qty > 0 )

ASSERTIONS
    /*pre origin*/
        o_key ∈ o_keys ∧ o_values ( o_key ) +
                        o_value ≥ 0 ∧
    /*promoted post*/
        p_result = p_id ∧
        p_catalog = p_catalog ∪ { p_id } ∧
        l_p_stock(p_id) = p_stock(p_id)+p_qty ∧
        l_p_stock ( p_id ) > p_stock ( p_id )
END
```

7.4.4 Interprétation des résultats de preuve et traçabilité

Après avoir généré les obligations de preuve, nous les analysons en utilisant le prouveur de l'AtelierB. La majorité d'entre elles ont été automatiquement prouvées. D'autres ont été prouvées manuellement en utilisant le prouveur interactif de l'AtelierB comme indiqué dans le tableau 7.4.1. Notons également qu'au départ, il y avait d'autres obligations non prouvées. Cela est dû au manque de typage de données au niveau des machines B. Nous avons donc corrigé et modifié les spécifications Kmelia pour pallier ce problème.

	Total	**Automatique**	**Manuel**
StockManager_obs	4	4	0
StockManager	12	9	3
Vendor_addItem	1	1	0
ve_addItem_sm_newReference	11	9	2
addToEntry_Store	6	4	2

Tableau 7.4.1 – Synthèse des obligations de preuve générées/prouvées avec l'AtelierB

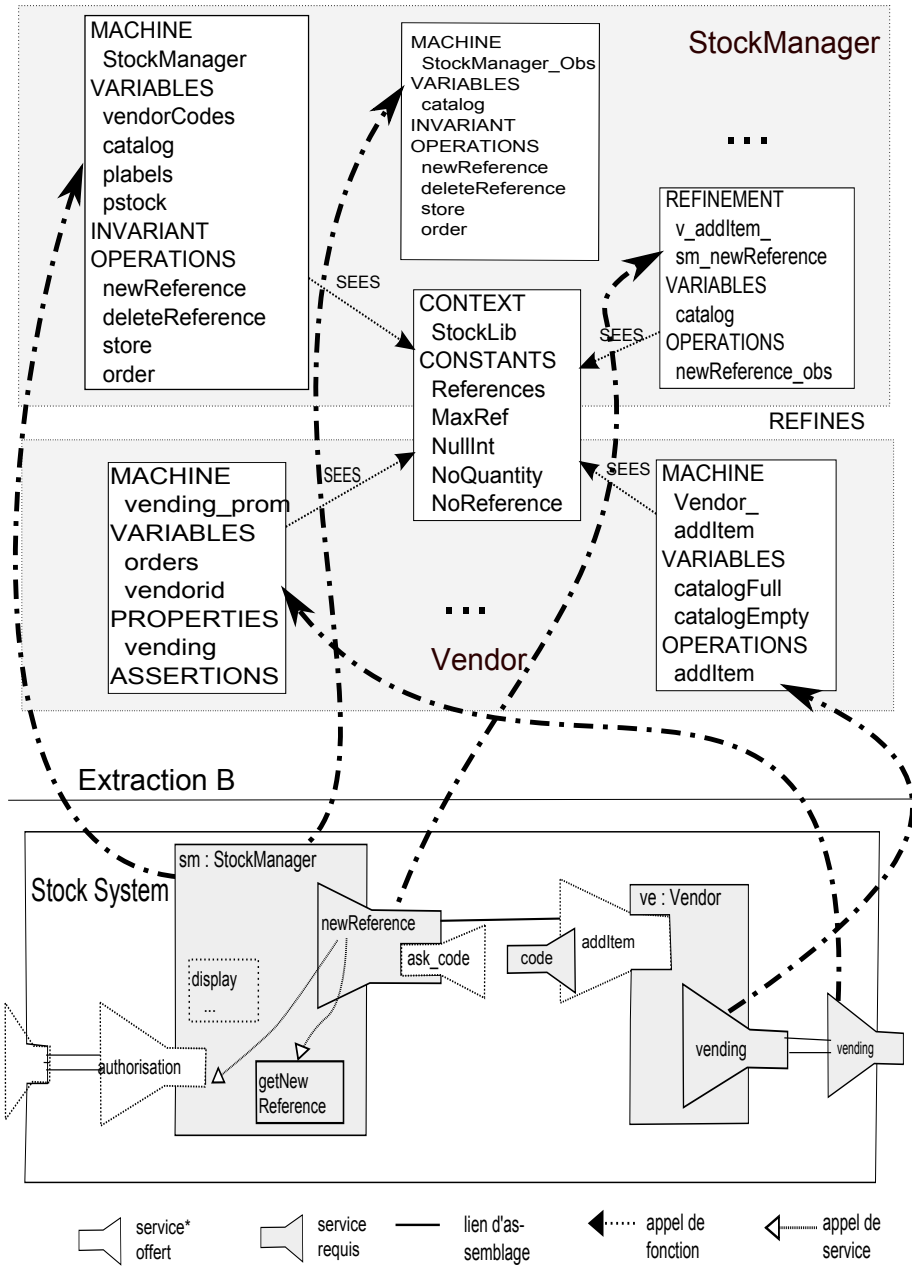


FIGURE 7.13 – Synthèse des machines B générées

L'interprétation des erreurs obtenues sur la spécification Kmelia de départ nous a été largement facilitée par la formalisation des règles de transformation. Considérons le message d'erreur $msg = (Inv_C, reference, Pred)$ que le prouveur de l'atelier B génère après échec de preuve d'une obligation de preuve, où Inv_C est un prédicat de l'invariant de la machine C en cours de vérification, $reference$ est une référence soit à une initialisation soit à une opération s et $Pred$ est un prédicat d'une substitution dans $reference$ qui ne respecte pas l'invariant. Nous améliorons la traçabilité des erreurs de spécification par une génération de messages d'erreurs $msg = (\mathbb{B}^{-1}[Inv_C], reference, \mathbb{B}^{-1}[Pred])$ sous la forme suivante :

"In a Kmelia component C, check that the invariant ($\mathbb{B}^{-1}[Inv_C]$) *is established by the service s"*
$=> \mathbb{B}^{-1}[Pred]$

Cette gestion des erreurs permet d'assister le spécifieur Kmelia et de rendre l'utilisation de B plus transparente.

7.5 Conclusion

Dans ce chapitre, nous avons présenté la plate-forme COSTO dédiée au modèle Kmelia. COSTO est basé sur Eclipse et intègre de nombreux plugins d'environnement et d'exportation. Les spécifications Kmelia sont les entrées de COSTO, elles sont d'abord chargées puis analysées. Les plugins d'exportation jouent le rôle de passerelles vers les outils externes.

Nous avons mis l'accent sur le prototype Kml2B qui présente les avantages suivants :

- il complète la liste de plugins d'exportation associée à COSTO ;
- il représente un maillon important pour rendre pragmatique la vérification des propriétés liées aux contrats ;
- son but est de contribuer à l'automatisation de l'approche que nous avons proposée dans le chapitre précédent. Ainsi, il extrait de façon automatique un modèle d'entrée (machines et raffinements) pour les outils associés à la méthode B.
- il permet de connecter la plate-forme COSTO aux différents outils de B (en particulier AtelierB).

Nous avons décrit l'architecture de ce plugin et son principe basé sur le patron de *visiteur*. Nous avons vu que ce dernier peut être facilement réutilisé pour développer d'autres plugins (par exemple vers Event-B, JML, Java, *etc.*). Enfin, nous avons illustré à la fois notre approche de vérification, l'utilisation du plugin Kml2B et les résultats d'analyse issus de l'AtelierB à travers l'exemple de gestion du stock.

Chapitre 8

Conclusion et perspectives

Ce chapitre dresse le bilan du travail réalisé au cours de cette thèse et présente les perspectives qui nous intéressent à court et à moyen terme.

8.1 Bilan

Les travaux de cette thèse ont eu pour objectif d'enrichir le *framework* Kmelia/COSTO afin de vérifier la conformité des spécifications d'un système à base de composants par rapport aux propriétés attendues dans le cahier des charges de ce dernier. Autrement dit, ce *framework* doit être capable de répondre aux questions liées aux aspects fonctionnels du système telles que :

- la *conformité d'un service* : le comportement de chaque service est-il conforme à sa spécification (ce que l'on attend de lui) ?
- la *cohérence d'un composant* : les propriétés du composant sont-elles préservées quel que soit son implantation ? Son protocole est-il cohérent ? Autrement dit, la manière dont il doit être utilisé est-elle réalisable ?
- la *correction d'un assemblage* de plusieurs composants : les composants sont-ils compatible ? Les composants respectent-ils leurs engagements et leurs garanties ?
- la *sûreté de la promotion* d'un service lorsqu'il est adapté à l'environnement du composite : le service promu préserve-t-il toujours son objectif fonctionnel ?

D'abord, nous avons abordé cette problématique en étudiant différentes propositions de modèles de composants représentatifs, dans l'objectif de ressortir les propriétés communes à prendre en considération lors de l'analyse d'un système à base de composants. Nous citons notamment les types d'interfaces exposées par les composants et les contrats associés (afin d'exprimer les propriétés à vérifier), le niveau d'abstraction et la composition des composants (afin de simplifier la vérification), le support de vérification et les outils associés (afin d'automatiser la vérification).

Cette étude nous a permis de constater que l'approche à composants est bâtie sur un cycle de développement qui fait une séparation claire entre les activités de développement de composants et celles de leur assemblage. Ces deux activités sont réalisées par des acteurs spécialisés qui peuvent ne pas être les mêmes. L'existence de ces deux activités explique diverses caractéristiques des composants, notamment

l'existence d'une vue externe qui donne la vision d'un composant en tant que boîte noire. C'est pourquoi le composant doit avoir des points d'accès clairement définis. En outre, afin de faciliter son utilisation (et sa réutilisation), il doit être décrit (spécifié, annoté, *etc.*) précisément pour permettre la compréhension de son fonctionnement sans avoir à accéder à son implantation interne. Pour répondre à ce besoin, nous avons étudié l'approche basée sur la notion de contrat dans le développement des logiciels.

En comparant les différents modèles existants, nous avons identifié trois principaux obstacles à la vérification des systèmes à base de composants :

- premièrement, il n'existe pas de modèle standard de composants. Ceci est principalement dû au fait que l'ensemble des acteurs concernés ont des priorités différentes concernant les caractéristiques devant être offertes par de tels modèles ;
- deuxièmement, la plupart des travaux autour de la vérification dans les modèles existants ne s'intéressent qu'à un sous-ensemble de la problématique de vérification (aspects structurels, aspects fonctionnels, ou aspects comportementaux). À notre connaissance, aucune approche n'a réellement traité la vérification des trois aspects d'une façon cohérente et intégrée (c'est-à-dire pas sous-forme d'extensions indépendantes) ;
- troisièmement, même dans le peu d'approches qui ont été proposées pour vérifier les systèmes à base de composants, cette vérification n'intervient malheureusement qu'après l'étape d'implantation (*runtime verification*).

Pour pallier ces obstacles, nous avons proposé, dans le cadre du modèle de composants Kmelia développé au sein de notre équipe, une approche de spécification et de vérification de logiciels à base de composants en utilisant la notion de contrat. Le tableau 8.1.1 montre une synthèse comparative issue de l'étude présentée dans le chapitre 2 ainsi qu'un positionnement de Kmelia par rapport aux différents modèles retenus. Nos principales contributions sont résumées dans les paragraphes suivants.

Un langage de données. Nous avons proposé un enrichissement du modèle Kmelia, à savoir un langage de données qui couvre la définition des types de données, des expressions, des assertions (pré/post-conditions, invariants, *etc.*), des communications, du typage des composants et des assemblages ainsi qu'une notion d'observabilité. Le pouvoir expressif de ce langage est nécessaire afin de satisfaire le double objectif fixé (spécification et vérification).

La prise en compte de données permet, à l'étape de spécification, d'exprimer des propriétés de cohérence plus riches au niveau des services (conformité du eLTS par rapport aux assertions pré/post-conditions), mais aussi au niveau des composants (préservation de l'invariant), des assemblages (le contrat de clientèle entre un service offert et requis) et de composition (encapsulation et sûreté de la promotion). L'avantage est que la vérification de telles propriétés est intégrée aux vérifications des autres propriétés sur les aspects structurels et dynamiques [AAM09, AAA+09].

Modèles	Formalismes	Composants		Contrats				Vérification				Génération de code	Plate-forme d'exécution
		Point d'accès	Composition	struct.	fonct.	comport.	QoS	Propriétés	Technique	Phase	Outils		
.NET	orienté programmation (Java, C#, Asml *etc.*)	opérations fournies	appel de méthode, non hiérarchique	oui	non[a]	non	non	conformité	simulation	*runtime*	Spec-Explorer	-	oui (.NET Framework)
EJB	orienté programmation (Java)	événement	appel de méthode, non hiérarchique	oui	non	non[b]	non	analyse statique, persistance	-	*runtime*	-	-	oui (JEE)
SOFA	abstrait (traces, behavior protocols)	interfaces fournies et requises (protocole)	assemblage, composite	oui	non	oui[b]	non	conformité des protocoles	inclusion de traces	*runtime*	-	non (squelettes)	oui (DCUP, SOFAnode)
Fractal	abstrait (Fractal-ADL, Fractal-IDL)	interfaces fournies et requises, interfaces de contrôle	assemblage, composite, partage	oui	oui[c]	non	non	typage d'interfaces, assertions (ConFract)	sous-typage	*runtime*	ConFract	-	Julia, Think (Java), Aokell(C/C++), Fract-Net(.NET)
Kmelia	formel (eLTS, logique)	interface (services offerts et requis)	assemblage, composite, service partagé, sous-services	oui	oui	oui	non	cf. tableaux 3.6.1 et 5.2.3	*model checking, theorem proving*	*design time*	COSTO	oui (partielle en Java)	non
UML 2.0	graphique	port/interface	assemblage, composite	oui	non[d] (OCL)	non[e]	non	-	-	-	-	non	non
Wright	formel (ADL/CSP)	interface avec sémantique précisée par le comportement	assemblage, composite[f]	oui	non	oui (CSP)	non	absence interblocage,	*model checking,* simulation	*design time*	Wr2fdr, FDR	non	non
Darwin	formel (ADL/CSP, π-calcul)	services fournis et requis	assemblage, composite	oui	non	oui (FSP)	non	analyse des scénarios (sûreté, vivacité)	*model checking*	*design time*	LTSA, SAA	oui (C++)	oui (Regis)
BIP	formel (RdP, LTS)	port	assemblage, composite	oui	non	oui (LTS)	non	protocole, détection d'interblocage	simulation	*design time*	IF-toolset	oui (C++)	oui (IF)

[a] une extension basée sur Eiffel a été proposée par ContractWizard.
[b] une extension basée sur les automates a été proposée dans CoCoNut/J.
[c] en utilisant CCL-J dans l'extension ConFract.
[d] les contraintes OCL peuvent être utilisées comme des contrats fonctionnels.
[e] les *statecharts* et les diagrammes d'activités peuvent être vus comme des contrats comportementaux.
[f] se fait par raffinement.

TABLE 8.1.1 – Synthèse comparative et positionnement de Kmelia.

Vérification des logiciels à base de composants en utilisant les contrats. Nous avons proposé des techniques de vérification pour prouver la préservation de l'intégrité des composants par les services, la correction fonctionnelle des services eux-mêmes et le respect des contrats d'assemblage et de promotion [MAA10c].

Ces techniques sont basées sur la notion de contrats multi-niveaux que nous avons proposée [MAA10a]. Les contrats multi-niveaux permettent, d'une part, d'expliciter la notion de responsabilité des différents éléments de modélisation (service, composant et assemblage, *etc.*) afin de maîtriser la construction progressive des systèmes à base de composants et, d'autre part, de maîtriser la complexité de la vérification de ces derniers.

L'originalité de cette approche réside dans le fait qu'elle intègre un processus de vérification modulaire : on commence par prouver que les services satisfont leurs contrats et on n'utilise ensuite que ces contrats, en abstrayant les services, pour prouver que, d'une part, le composant les contenant est cohérent et, d'autre part, que les composants, mis ensemble à travers ses services, satisfont bien les propriétés qu'on leur a associées.

Mise en œuvre de l'approche. Nous avons montré comment vérifier la cohérence des composants, des assemblages et des compositions Kmelia en se servant des outils de preuve dédiés à la méthode B. La transposition de la vérification a été rendue possible par l'adoption d'une approche de vérification basée sur des contrats pré/post-conditions (aussi bien pour les services des composants que pour leur assemblage et leur composition) qui s'avèrent être compatibles avec les obligations de preuve de cohérence des spécifications B. Nous extrayons alors systématiquement, sur la base de règles de traduction que nous avons formalisées, des spécifications B à partir des composants ou assemblages Kmelia [MAA10b, AAM10]. Les machines B résultant de cette extraction sont fournies à l'AtelierB et soumises à la preuve.

Les expérimentations que nous avons menées ont montré que les erreurs dans les spécifications Kmelia se traduisent par des obligations de preuve non déchargées. Elles peuvent alors être corrigées et vérifiées à nouveau.

Implantation. Nous avons développé, comme extension de la plate-forme COSTO, un nouveau plugin appelé Kml2B qui complète la liste des plugins d'exportation associée à COSTO. Le plugin Kml2B représente un maillon important pour rendre pragmatique la vérification des propriétés liées aux contrats. Son but est d'extraire un modèle d'entrée (machines et raffinements) pour les outils associés à la méthode B. La plate-forme COSTO est donc désormais connectée avec les différents outils de B (en particulier l'AtelierB) [AMAA11, AAM11]. Nous avons illustré notre approche à travers l'exemple de gestion du stock.

8.2 Perspectives

Nous présentons maintenant quelques points que nous souhaiterions traiter dans des travaux futurs. Nous commençons par deux problématiques déjà en cours d'expérimentation, puis nous exposons nos perspectives à plus long terme.

Diagnostic et rétroaction des résultats de l'analyse. Un travail entrepris pour le court terme est l'aide à la rétroaction sur les spécifications Kmelia (cf. section 7.4.4) ; il s'agit d'exploiter les résultats des analyses en B pour référencer les parties erronées dans la spécification Kmelia de départ. La proximité des structures de Kmelia avec celles de B et les règles de traduction définies devraient aider à obtenir rapidement des résultats dans ce sens.

Notons que le processus du diagnostic est déclenché lorsque la spécification B ne peut être prouvée, c'est-à-dire s'il existe au moins une obligation de preuve (cf. section 6.1.3) non déchargée par le prouveur. La grande difficulté est de savoir si cet échec est dû à une erreur dans la spécification ou à une incapacité de la part du prouveur automatique ou de l'utilisateur humain (prouveur interactif) à mener à bien la preuve.

Correction fonctionnelle des services. Une autre piste que nous explorons actuellement concerne la correction fonctionnelle des services. Elle permet de compléter les propriétés prouvées sur les spécifications Kmelia dans le but d'obtenir des composants et des assemblages corrects. Nous voulons prouver que le comportement décrit sous forme de eLTS (cf. la section 3.4.2) associé au service est en adéquation avec les pré/post-conditions, c'est-à-dire qu'en supposant la pré-condition, le comportement établit la post-condition.

Parmi les pistes que nous explorons, il y a également le calcul de la pré-condition en partant de la post-condition et en remontant à l'état initial de l'automate qui décrit le système. Comme nous l'avons déjà évoqué dans la section 5.3.1, la correction fonctionnelle appliquée aux eLTS est un problème difficile à résoudre dans sa globalité, du fait de la non-structuration des comportements et de la présence de boucles et de types de données avancés. Une solution est de commencer par caractériser un sous-ensemble des eLTS considérés et limiter les types de données manipulées. En effet, une boucle dans un eLTS ne peut pas forcément être transformée en une structure de contrôle comparable aux boucles que nous trouvons dans la plupart des langages de programmation. Même pour ces dernières, ni en théorie, ni en pratique, il n'y a pas une façon générale pour calculer leur plus faible pré-condition.

Une approche a été proposée pour pallier ce problème. Elle consiste à utiliser des invariants de boucle qui peuvent être fournis par le spécifieur ou par un algorithme de génération d'invariants. Nous explorons aussi une autre piste qui consiste à traduire le comportement du service en Event-B puis à prouver la cohérence du modèle Event-B généré par rapport à l'invariant (modélisant la post-condition) quand le déroulement du service atteint un état final.

Raffinement vers des plates-formes d'exécution. Un des objectifs initiaux de Kmelia est de construire des composants corrects par raffinement des spécifications abstraites. Nous aimerions travailler sur le raffinement des spécifications Kmelia vers des applications opérationnelles en nous appuyant sur des plates-formes spécifiques (*middleware*) offrant les possibilités d'interaction nécessaires. Quelques expérimentations préliminaires avec Java ont déjà été commencées. Certaines approches comme Sofa ou Fractal plaquent l'architecture à composants au-dessus du code, la généra-

tion de code de l'architecture vient ainsi compléter le code (métier). Pour pouvoir vérifier les propriétés fonctionnelles liées aux expressions et aux assertions, Kmelia est un langage à large spectre. De ce fait, le raffinement peut se faire progressivement, à condition de prouver les étapes, ou directement par génération de code selon le niveau d'abstraction des spécifications. Nous pensons qu'une cible intéressante pour atteindre cet objectif est d'exploiter les technologies fournies par le modèle SCA [1] et notamment la proximité de ces concepts à ceux du modèle Kmelia. Notons que les résultats liés au raffinement des spécifications Kmelia nous permettraient également d'envisager la prise en comptes des propriétés non-fonctionnelles. Puisque ces dernières ne sont pas encore traitées car elles sont fortement liées aux phases de déploiement et d'exécution. Nous pouvons citer, par exemple, les propriétés liées à à la tolérance aux fautes dans le cadre des systèmes embarqués.

Enfin, les perspectives susmentionnées permettraient la création d'un environnement de développement plus complet où toutes les vues du système en cours de développement seraient modifiables indépendamment les unes des autres. Cet environnement serait capable de combiner les différentes vues dans un cadre cohérent afin générer une implantation correcte vis-à-vis de chacune d'elles.

1. http ://www.osoa.org/display/Main/Service+Component+Architecture+Home

Bibliographie

[AAA+04] P. André, G. Ardourel, C. Attiogbé, H. Habrias et C. Stoquer : Vérification de conformité des interactions entre composants. Rapport technique RR04.11, LINA - FRE CNRS 2729 - Nantes, December 2004.

[AAA06a] Pascal André, Gilles Ardourel et Christian Attiogbé : Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition. *In 1ère Conférence Francophone sur les Architectures Logicielles*, pages 101–118. Hermès, Lavoisier, 2006.

[AAA06b] Pascal André, Gilles Ardourel et Christian Attiogbé : Vérification d'assemblage de composants logiciels Expérimentations avec MEC. *In* Michel Gourgand et Fouad Riane, éditeurs : *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006*, pages 497–506, Rabat, Maroc, avril 2006. Lavoisier.

[AAA06c] Christian Attiogbé, Pascal André et Gilles Ardourel : Checking Component Composability. *In 5th International Symposium on Software Composition, SC'06*, volume 4089 de *LNCS*. Springer, 2006.

[AAA07a] Pascal André, Gilles Ardourel et Christian Attiogbé : A Formal Analysis Toolbox for the Kmelia Component Model. *In Proceedings of ProVeCS'07 (TOOLS Europe)*, numéro 567 de Technichal Report. ETH Zurich, 2007.

[AAA07b] Pascal André, Gilles Ardourel et Christian Attiogbé : Defining Component Protocols with Service Composition : Illustration with the Kmelia Model. *In 6th International Symposium on Software Composition, SC'07*, volume 4829 de *LNCS*. Springer, 2007.

[AAA08] Pascal André, Gilles Ardourel et Christian Attiogbé : Composing Components with Shared Services in the Kmelia Model. *In 7th International Symposium on Software Composition, SC'08*, volume 4954 de *LNCS*. Springer, 2008.

[AAA+09] Pascal André, Gilles Ardourel, Christian Attiogbé, Arnaud Lanoix et Mohamed Messabihi : Prise en compte d'assertions pour la correction d'assemblages de composants Kmelia. *In Journée GDR-GPL (COSMAL)*, Mai, 2009.

[AAA11] Pascal André, Gilles Ardourel et Christian Attiogbé : Kmelia : un modèle abstrait et formel pour la description et la composition de

composants et de services. *Technique et Science Informatique (TSI)*, 30, 2011.

[AAM09] Pascal André, Christian Attiogbé et Messabihi Mohamed : Correction d'assemblages de composants impliquant des interfaces paramétrées. *In 3e Conférence Francophone sur les Architectures Logicielles*, volume RNTI-L-4 de *Revue des Nouvelles Technologies de l'Information*, pages 33–44. Cépaduès-Éditions, 2009.

[AAM10] Pascal André, Gilles Ardourel et Mohamed Messabihi : Component service promotion : Contracts, mechanisms and safety. *Electronic Notes in Theoretical Computer Science*, 2010. Proceedings of the 7th International Workshop on Formal Aspects of Component Software (FACS 2010).

[AAM11] Pascal André, Gilles Ardourel et Mohamed Messabihi : Vérification de contrats logiciels à l'aide de transformations de modèles : Application à Kmelia. *In 7ème journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, jui, 2011.

[ABHV06] J.-R. Abrial, M. Butler, S. Hallerstede et L. Voisin : An Open Extensible Tool Environment for Event-B. *In ICFEM 2006*, volume 4260 de *LNCS*. Springer, 2006.

[Abr96] Jean-Raymond Abrial : *The B-Book Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[Abr10] Jean-Raymond Abrial : *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[AC01] Giuseppe Attardi et Antonio Cisternino : Reflection support by means of template metaprogramming. *In Proceedings of Third International Conference on Generative and Component-Based Software Engineering, LNCS*, pages 118–127. Springer-Verlag, 2001.

[ACN02a] Jonathan Aldrich, Craig Chambers et David Notkin : Architectural reasoning in ArchJava. *In Proceedings ECOOP 2002*, volume 2374 de *LNCS*, pages 334–367. Springer Verlag, 2002.

[ACN02b] Jonathan Aldrich, Craig Chambers et David Notkin : Archjava : connecting software architecture to implementation. *In ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[AG97] Robert Allen et David Garlan : A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[AGA+08] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy et K. Holley : Soma : a method for developing service-oriented solutions. *IBM Syst. J.*, 47:377–396, July 2008.

[AH07a] J.-R. Abrial et S. Hallerstede : Refinement, Decomposition, and Instantiation of Discrete Models : Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[AH07b] J.-R. Abrial et S. Hallerstede : Refinement, Decomposition, and Instantiation of Discrete Models : Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[AL03] P. Attie et D. H. Lorenz : Correctness of Model-based Component Composition without State Explosion. *In ECOOP 2003 Workshop on Correctness of Model-based Software Composition*, 2003.

[All97] Robert Allen : *A Formal Approach to Software Architecture*. Thèse de doctorat, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[AMAA11] Pascal André, Mohamed Messabihi, Gilles Ardourel et Christian Attiogbé : COSTO/Kmelia : a Platform to Specify and Verify Component and Service Software . *In Journée GDR-GPL (COSMAL)*, Juin, 2011.

[APRS01] Colin Atkinson, Barbara Paech, Jens Reinhold et Torsten Sander : Developing and applying component-based model-driven architectures in kobra. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:0212, 2001.

[AS01a] Karine Arnout et Raphaël Simon : The .net contract wizard : Adding design by contract to languages other than eiffel. *In Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, TOOLS '01, pages 14–, Washington, DC, USA, 2001. IEEE Computer Society.

[AS01b] Karine Arnout et Raphaël Simon : The .net contract wizard : Adding design by contract to languages other than eiffel. *In TOOLS '01 : Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, page 14, Washington, DC, USA, 2001. IEEE Computer Society.

[Att08] Christian Attiogbé : Mastering Specification Heterogeneity with Multifacet Analysis. *In MoVaH'2008 - ICST'08 Workshop*, IEEE, 2008.

[Bak97] Sean Baker : *CORBA - Distributed Objects, The Orbix Approach*. ACM Press. Addison Wesley, 1997. ISBN 0-201-92475-7.

[BB87] Tommaso Bolognesi et Ed Brinksma : Introduction to the iso specification language lotos. *Computer Networks*, 14:25–59, 1987.

[BBC+98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang et W. Weiss : An architecture for differentiated service, 1998.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino et Erik Poll : An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[BCFS05] Jean-Paul Bodeveix, David Chemouil, Mamoun Filali et Martin Strecker : Towards formalising aadl in proof assistants. *Electr. Notes Theor. Comput. Sci.*, 141(3):153–169, 2005.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma et Jean-Bernard Stefani : An open component model and its support in java. *In CBSE*, pages 7–22, 2004.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma et J.-B. Stefani : The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12), 2006.

[BCMR07] Tomás Barros, Antonio Cansado, Eric Madelaine et Marcela Rivera : Model-checking distributed components : The vercors platform. *Electron. Notes Theor. Comput. Sci.*, 182:3–16, 2007.

[BDH+08] Tomáš Bureš, Martin Děcký, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý et Petr Tůma : *CoCoME in SOFA*, pages 1–2. Volume 5153 de Rausch *et al.* [RRMP08], 2008.

[Ber96] Philip A. Bernstein : Middleware : a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[BGM02] Marius Bozga, Susanne Graf et Laurent Mounier : If-2.0 : A validation environment for component-based real-time systems. *In In Proceedings of Conference on Computer Aided Verification, CAV'02*, pages 343–348. Springer Verlag, 2002.

[BH06] Peter F Brown et Rebekah Metz Booz Allen Hamilton : Reference model for service oriented architecture 1.0, 2006.

[BHH+06] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp et Martin Wirsing : A component model for architectural programming. *Electr. Notes Theor. Comput. Sci.*, 160:75–96, 2006.

[BHP06] Tomas Bures, Petr Hnetynka et Frantisek Plasil : Sofa 2.0 : Balancing advanced features in a hierarchical component model. *In Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BJPW99a] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau et Damien Watkins : Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BJPW99b] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau et Damien Watkins : Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BS03] Mike Barnett et Wolfram Schulte : Runtime verification of .net contracts. *J. Syst. Softw.*, 65(3):199–208, 2003.

[BSC99] Preeti Bhoj, Sharad Singhal et Sailesh Chutani : Sla management in federated environments. *In Computer Networks*, pages 293–308. IEEE Publishing, 1999.

[BZ08] Mario Bravetti et Gianluigi Zavattaro : A foundational theory of contracts for multi-party service composition. *Fundam. Inf.*, 89:451–478, December 2008.

[Car03] Eric Cariou : *Contribution à un processus de réification d'abstractions de communication.* Thèse de doctorat, Université de Rennes 1, juillet 2003.

[CCL06] Ivica Crnkovic, Michel Chaudron et Stig Larsson : Component-based development process and component lifecycle. *In Proceedings of the International Conference on Software Engineering Advances*, pages 44–, Washington, DC, USA, 2006. IEEE Computer Society.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith et Sanjiva Weerawarana : Web service definition language (wsdl). Rapport technique NOTE-wsdl-20010315, World Wide Web Consortium, March 2001.

[CFN03a] Cyril Carrez, Alessandro Fantechi et Elie Najm : Behavioural contracts for a sound composition of components. *In* Hartmut König, Monika Heiner et Adam Wolisz, éditeurs : *FORTE 2003, IFIP TC6/WG 6.1)*, volume 2767 de *LNCS*, pages 111–126. Springer-Verlag, Berlin, Germany, septembre 2003.

[CFN03b] Cyril Carrez, Alessandro Fantechi et Elie Najm : Contrats comportementaux pour un assemblage sain de composants. *In Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2003)*, Paris, France, octobre 2003. Traduction française de [CFN03a].

[CFP+03] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M. Troya et Antonio Vallecillo : Adding Roles to CORBA Objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.

[CFS03] Christophe Calandreau, Alain Fauré et Nader Soukouti : *EJB 2.0 - Mise en oeuvre.* Dunod, 2003. ISBN 2-10-004729-9.

[CG87] Edmund M. Clarke et Orna Grumberg : The model checking problem for concurrent systems with many similar processes. *In Temporal Logic in Specification*, pages 188–201, 1987.

[CH01] Bill Councill et George T. Heineman : Definition of a software component and its elements. pages 5–19, 2001.

[CL00] Ivica Crnkovic et Magnus Larsson : Component based software engineering - state of the art. Rapport technique, January 2000.

[CQSS07] T. Coupaye, V. Quéma, L. Seinturier et J.-B. Stefani : *Intergiciel et Construction d'Applications Réparties*, chapitre Le système de composants Fractal. InriAlpes, janvier 2007. sardes.inrialpes.fr/ecole/livre/pub/.

[CR05] Philippe Collet et Roger Rousseau : ConFract : un système pour contractualiser des composants logiciels hiérarchiques. *L'Objet, LMO'05*, 11(1-2):223–238, 2005.

[CRCR05] P. Collet, R. Rousseau, T. Coupaye et N. Rivierre : A Contracting System for Hierarchical Components. *In* George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski et Kurt C. Wallnau, éditeurs : *CBSE*, volume 3489 de *Lecture Notes in Computer Science*, pages 187–202. Springer, 2005.

[CS06] Thierry Coupaye et Jean-Bernard Stefani : Fractal component-based software engineering. *In* Mario Südholt et Charles Consel, éditeurs : *ECOOP Workshops*, volume 4379 de *Lecture Notes in Computer Science*, pages 117–129. Springer, 2006.

[dAH01] Luca de Alfaro et Thomas A. Henzinger : Interface automata. *In ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[DCL08] Zuohua Ding, Zhenbang Chen et Jing Liu : A rigorous model of service component architecture. *Electr. Notes Theor. Comput. Sci.*, 207:33–48, 2008.

[Dij75] Edsger W. Dijkstra : Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

[Dij78] Edsger W. Dijkstra : Finding the correctness proof of a concurrent program. *In MFCS'78*, pages 31–38, 1978.

[DW99] Desmond F. D'Souza et Alan Cameron Wills : *Objects, components, and frameworks with UML : the catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Edm99] Bruce Edmonds : The pragmatic roots of context. *In Proceedings of the Second International and Interdisciplinary Conference on Modeling and Using Context*, CONTEXT '99, pages 119–132, London, UK, 1999. Springer-Verlag.

[EFLA04] Daniel Enselme, Gerard Florin et Fabrice Legond-Aubry : Design by contract : analysis of hidden dependencies in component based application. *Journal of Object Technology*, 3(4):23–45, 2004.

[Erl05] Thomas Erl, éditeur. *Service-Oriented Architecture - Concepts, Technology, and Design.* Prentice Hall, Boston, MA, USA, 2005.

[Flo67] R. W. Floyd : Assigning meanings to programs. *In* J. T. Schwartz, éditeur : *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[Fro02] Ákos Frohner, éditeur. *Object-Oriented Technology ECOOP 2001 Workshop Reader, ECOOP 2001 Workshops, Panel, and Posters, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2323 de *Lecture Notes in Computer Science*. Springer, 2002.

[FW07] Leo Freitas et Jim Woodcock : Fdr explorer. *Electron. Notes Theor. Comput. Sci.*, 187:19–34, July 2007.

[GA94] D. Garlan et R Allen : Formalizing Architectural Connection. *In Proceedings of the 16th ICSE*, pages 71–80. IEEE Computer Society Press, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides : *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri et Dino Mandrioli : *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd édition, 2002.

[Gri89] Alain Griffault : *Contribution à l'étude des systèmes communicants et des algorithmes d'exclusion mutuelle*. Thèse de doctorat, Université de Bordeaux I, Janvier 1989.

[Gro03] Object Management Group : UML2.0 Superstructure Specification : Final Adopted Specification. Rapport technique, http ://www.omg.org/docs/ptc/03-08-02.pdf, August 2003.

[GS05] Gregor Gossler et Joseph Sifakis : Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[HC01] George T. Heineman et William T. Councill, éditeurs. *Component-based software engineering : putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hoa69] C. A. R. Hoare : An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa83] C. A. R. Hoare : Communicating sequential processes. *Commun. ACM*, 26:100–106, January 1983.

[Hol03] Gerard Holzmann : *Spin model checker, the : primer and reference manual*. Addison-Wesley Professional, first édition, 2003.

[Hor96] James Horning : The larch shared language : Some open problems. *In In M. Haveraaen, O. Owe, O.-J. Dahl (Eds.) : Recent Trends in Data Type Specification. LNCS 1130*, pages 58–73. Springer Verlag, 1996.

[ISO98] ISO/IEC : *quality of service : Framework*. Itu-t recommandation x.641 - iso/iec 13236 : Information technology, 1998.

[Jau94] Patrick Jaulent : *Génie logiciel, les méthodes*. 9782200214821. Armand Colin, 06 1994.

[JM97] Jean-Marc Jézéquel et Bertrand Meyer : Design by contract : The lessons of ariane. *Computer*, 30:129–130, January 1997.

[KAB+06] J. Kofron, J. Adamek, T. Bures, P. Jezek, V. Mencl, P. Parizek et F. Plasil : Checking fractal component behavior using behavior protocols. presented at the 5th fractal workshop (part of ecoop'06, 2006.

[KL03] Er Keller et Heiko Ludwig : The wsla framework : Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:2003, 2003.

[KL04] Olga KOUCHNARENKO et Arnaud LANOIX : Verifying invariants of component-based systems through refinement. *In AMAST*, pages 289–303, 2004.

[Kra98] Reto KRAMER : icontract - the java(tm) design by contract(tm) tool. *In In TOOLS '98 : Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295. IEEE Computer Society, 1998.

[Krä08] Bernd J. KRÄMER : Component meets service : what does the mongrel look like ? *ISSE*, 4(4):385–394, 2008.

[Küh06] Thomas KÜHNE : Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[LDK04] Heiko LUDWIG, Asit DAN et Robert KEARNEY : Cremona : An architecture and library for creation and monitoring of ws-agreents. *In ICSOC*, pages 65–74, 2004.

[LRR+00] Gary T. LEAVENS, Clyde RUBY, K. RUSTAN, M. LEINO, Erik POLL et Bart JACOBS : Jml (poster session) : notations and tools supporting detailed design in java. *In OOPSLA '00 : Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM.

[LW07] K.-K. LAU et Z. WANG : Software component models. *IEEE Trans. on Software Engineering*, 33(10):709–724, October 2007.

[MAA10a] Mohamed MESSABIHI, Pascal ANDRÉ et Christian ATTIOGBÉ : Multilevel contracts for trusted components. *In WCSI*, pages 71–85, 2010.

[MAA10b] Mohamed MESSABIHI, Pascal ANDRÉ et Christian ATTIOGBÉ : Preuve de cohérence de composants Kmelia à l'aide de la méthode B. *In 4ème Conférence Francophone sur les Architectures Logicielles*, volume L-4 de *RNTI*, pages 113–126. Cépaduès-Éditions, 2010.

[MAA10c] Mohamed MESSABIHI, Pascal ANDRÉ et Christian ATTIOGBÉ : Correction d'assemblages de composants impliquant des données et assertions. *In Journée GDR-GPL (COSMAL)*, Mars, 2010.

[Mag99] Jeff MAGEE : Behavioral analysis of software architectures using ltsa. *In ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 634–637, New York, NY, USA, 1999. ACM.

[Mey92] Bertrand MEYER : Applying "design by contract". *Computer*, 25(10): 40–51, 1992.

[Mey97] B. MEYER : *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2nd édition, 1997. `http://www.eiffel.com/doc/oosc/`.

[Mey03] B. MEYER : The Grand Challenge of Trusted Components. *In Proceedings of 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003.

[Mil82] R. MILNER : *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil98] Robin MILNER : The pi calculus and its applications (keynote address). *In IJCSLP*, pages 3–4, 1998.

[MKG99a] Jeff MAGEE, Jeff KRAMER et Dimitra GIANNAKOPOULOU : Behaviour analysis of software architectures. *In WICSA1 : Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[MKG99b] Jeff MAGEE, Jeff KRAMER et 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, 1999. Kluwer, B.V.

[Mor05] Lionel MOREL : *Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille.* Thèse de doctorat, Institut National Polytechnique de Grenoble, 2005.

[MP06] Vladimir MENCL et Matej POLAK : M. : Uml 2.0 components and fractal : An analysis. *In Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06*, 2006.

[MT00] N. MEDVIDOVIC et R. N. TAYLOR : A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, january 2000.

[ND95] Oscar NIERSTRASZ et Laurent DAMI : Component-oriented software technology, 1995.

[OKS05a] M. OUSSALAH, T. KHAMMACI et A. SMEDA : *Les composants : définitions et concepts de base*, chapitre 1, pages 1–18. Les systèmes à base de composants : principes et fondements. M. Oussalah et al. Eds, Editions Vuibert, 2005.

[OKS05b] M. OUSSALAH, T. KHAMMACI et A. SMEDA : *Les composants : définitions et concepts de base*, chapitre 1, pages 1–18. Les systèmes à base de composants : principes et fondements. M. Oussalah et al. Eds, Editions Vuibert, 2005.

[OMG05] OMG : *Object Constraint Language Specification, version 2.0.* Object Modeling Group, June 2005.

[Oza07a] Alain OZANNE : *Interact : un modèle général de contrat pour la garantie des assemblage de composants et services.* Thèse de doctorat, Université Pierre et Marie Curie (Paris VI), November 2007.

[OZA07b] Alain OZANNE : *Interact : un modèle général de contrat pour la garantie des assemblages de composants et services.* Thèse de doctorat, L'Université Pierre & Marie Curie - Paris VI, Novembre 2007.

[Pap03] M. P. PAPAZOGLOU : Service-Oriented Computing : Concepts, Characteristics and Directions. *In WISE*, pages 3–12. IEEE Computer Society, 2003.

[PK02] Reinhold PLÖSCH et Johannes KEPLER : Evaluation of assertion support for the java programming language. *Journal of Object Technology*, 1:2002, 2002.

[PNPR05] Sebastian PAVEL, Jacques NOYE, Pascal POIZAT et Jean-Claude ROYER : Java Implementation of a Component Model with Explicit Symbolic Protocols. *In 4th International Symposium on Software Composition, SC'05*, volume 3628 de *LNCS*. Springer, 2005.

[PV02] F. PLASIL et S. VISNOVSKY : Behavior protocols for software components, 2002. IEEE Transactions on SW Engineering, 28 (9), 2002.

[QT06] A. QUERALT et E. TENIENTE : Reasoning on UML Class Diagrams with OCL Constraints. *In ER*, volume 4215 de *LNCS*, pages 497–512. Springer, 2006.

[Rap02] RAPHAELMARVIE : *Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants.* Thèse de doctorat, Université des Sciences et Techniques de Lille, LIFL, Villeneuve d'Ascq, Décembre 2002.

[Reu03] Ralf H. REUSSNER : Automatic component protocol adaptation with the coconut/j tool suite. *Future Gener. Comput. Syst.*, 19:627–639, July 2003.

[RRMP08] Andreas RAUSCH, Ralf REUSSNER, Raffaela MIRANDOLA et Frantisek PLASIL, éditeurs. *The Common Component Modeling Example : Comparing Software Component Models*, volume 5153 de *LNCS*. Springer, Heidelberg, 2008.

[Rua84] Lawrence M. RUANE : Abstract data types in assembly language programming. *SIGPLAN Not.*, 19(1):63–67, 1984.

[SC00] ao Costa SECO, Jo et Luís CAIRES : A basic model of typed components. *In ECOOP '00 : Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, UK, 2000. Springer-Verlag.

[Sch03] H. SCHMIDT : Trustworthy components-compositionality and prediction. *J. Syst. Softw.*, 65(3):215–225, 2003.

[SDW93] Ketil STØLEN, Frank DEDERICHS et Rainer WEBER : Assumption/commitment rules for networks of asynchronously communicating agents. Rapport technique, 1993.

[Sif05] Joseph SIFAKIS : A Framework for Component-based Construction Extended Abstract. *In SEFM'05 : Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 293–300, Washington, DC, USA, 2005. IEEE Computer Society.

[Spi89] J. M. SPIVEY : *The Z notation : a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Szy97] Clemens SZYPERSKI : *Component Software : Beyond Object-Oriented Programming.* AddisonWesley Publishing Company, 1997.

[Szy02] Clemens SZYPERSKI : *Component Software : Beyond Object-Oriented Programming*. Addison Wesley Publishing Company/ACM Press, 2002. ISBN 0-201-74572-0.

[TEG+97] Dean THOMPSON, Chris EXTON, Leah GARRETT, A.S.M. SAJEEV et Damien WATKINS : Distributed component object model (dcom), 1997.

[Tho06] Dave THOMAS, éditeur. *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 de *Lecture Notes in Computer Science*. Springer, 2006.

[Tou05] Jean-Charles TOURNIER : *Qinna, une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles*. Thèse de doctorat, INSA de Lyon, 07 2005. Jean-Bernard StéfaniJean-Marc JézéquelJean-Marc Geib.

[Vin97] Steve VINOSKI : Corba : Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, pages 35–2, 1997.

[VTS02] G. J. VECELLIO, W. M. THOMAS et R. M. SANDERS : Containers for predictable behavior of component-based software. *In Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, 2002.

[Win90] Jeannette M. WING : A specifier's introduction to formal methods. *Computer*, 23:8–23, September 1990.

[YS97] D.M. YELLIN et R.E. STROM : Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

Liste des publications

[AAM09] Pascal André , Christian Attiogbé , Mohamed Messabihi. **Correction d'assemblages de composants impliquant des interfaces paramétrées**. In 3ème Conférence Francophone sur les Architectures Logicielles, RNTI-L-4 of *Revue des Nouvelles Technologies de l'Information*, pages 33-44. Cépaduès-Éditions, 2009.

[AAA+09] Pascal André, Gilles Ardourel, Christian Attiogbé, Arnaud Lanoix, Mohamed Messabihi. **Prise en compte d'assertions pour la correction d'assemblages de composants Kmelia**. Communication dans la journée GDR-GPL (COSMAL), Toulouse, France, Mai, 2009.

[MAA10a] Mohamed Messabihi, Pascal André, Christian Attiogbé. **Multi-levels Use of Contracts for Trusted Components**. In WCSI TOOLS'2010. *Electronic Proceedings in Theoretical Computer Science*, pages 85-100. Malaga, Spain, Juin, 2010.

[MAA10b] Mohamed Messabihi, Pascal André, Christian Attiogbé. **Preuve de cohérence des composants Kmelia à l'aide de la méthode B**. In 4ème Conférence Francophone sur les Architectures Logicielles, RNTI-L-4 of *Revue des Nouvelles Technologies de l'Information*, pages 111-125. Cépaduès-Éditions, 2010.

[MMA10c] Mohamed Messabihi, Pascal André, Christian Attiogbé. **Correction d'assemblages de composants impliquant des données et assertions**. *Communication invitée dans la journée GDR-GPL*, Pau, France, Mars 2010.

[AAM10] Pascal André, Gilles Ardourel, Mohamed Messabihi **Component Service Promotion : Contracts, Mechanisms and Safety** *In 7th International Workshop on Formal Aspects of Component Software (FACS 2010)*, Guimarães, Portugal, October, 2010.

[AAM11] Pascal André, Gilles Ardourel, Mohamed Messabihi **Vérification de contrats logiciels à l'aide de transformations de modèles : Application à Kmelia**. In : *7ème journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, Lille, France. Juin, 2011.

[AMAA11] Pascal André, Mohamed Messabihi, Gilles Ardourel, Christian Attiogbé. **COSTO/Kmelia : a Platform to Specify and Verify Component and Service Software**. *Poster soumis dans la journée GDR-GPL (COSMAL)*, Lille, France, Juin 2011.

Résumé

L'utilisation croissante des composants et des services logiciels dans les différents secteurs d'activité (télécommunications, transports, énergie, finance, santé, *etc.*) exige des moyens (modèles, méthodes, outils, *etc.*) rigoureux afin de maîtriser leur production et d'évaluer leur qualité. En particulier, il est crucial de pouvoir garantir leur bon fonctionnement en amont de leur déploiement lors du développement modulaire de systèmes logiciels.

Kmelia est un modèle à composants multi-services développé dans le but de construire des composants logiciels et des assemblages prouvés corrects. Trois objectifs principaux sont visés dans cette thèse. Le premier consiste à enrichir le pouvoir d'expression du modèle Kmelia avec un langage de données afin de satisfaire le double besoin de spécification et de vérification. Le deuxième vise l'élaboration d'un cadre de développement fondé sur la notion de contrats multi-niveaux. L'intérêt de tels contrats est de maîtriser la construction progressive des systèmes à base de composants et d'automatiser le processus de leur vérification. Nous nous focalisons dans cette thèse sur la vérification des contrats fonctionnels en utilisant la méthode B. Le troisième objectif est l'instrumentation de notre approche dans la plate-forme COSTO/Kmelia. Nous avons implanté un prototype permettant de connecter COSTO aux différents outils associés à la méthode B. Ce prototype permet de construire les machines B à partir des spécifications Kmelia en fonction des propriétés à vérifier. Nous montrons que la preuve des spécifications B générées garantit la cohérence des spécifications Kmelia de départ. Les illustrations basées sur l'exemple CoCoME confortent nos propositions.

Mots-clés : Modèle de composants, Services, Contrats, Vérification formelle, Méthode B.

Abstract

The pervasiveness of software components and services in various domains (telecommunications, transportation, energy, financial transactions, health, *etc.*) requires rigorous means (models, methods, tools, *etc.*) to control their production and to assess their quality. In particular, when developing such modular systems, it is crucial to ensure their safe behaviour before deployment.

Kmelia is a multi-services component model developed with the aim of building correct software components and assemblies. Three main goals are covered in this thesis. The first one is to enrich the expressiveness of the Kmelia model with a data language in order to satisfy the twofold need of specification and verification. The second one is to provide a framework development based on the concept of multi-level contracts. The interest of such contracts is to master the incremental construction of component-based systems and to automate the process of their verification. In this thesis, we focus on the verification of functional contracts using the B method. The last goal is to implement our approach in the COSTO/Kmelia platform. We developed a prototype that connects COSTO to various B method tools. This prototype builds B machines from Kmelia specifications according to the set of properties to check. We show that proof of the generated B specifications ensures consistency of initial Kmelia specifications. Several examples from the CoCoME case study consolidate our proposal.

Keywords : Component model, Services, Contrats, Formal verification, B method.