THÈSE no 2309

Configuration Systématique de Middleware

présentée

DEVANT L' UNIVERSITÉ DE RENNES 1

pour obtenir le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

PAR

Apostolos ZARRAS

COMPOSITION DU JURY:

Gul Agha

Jean-Pierre Banâtre

Gordon Blair

Valérie Issarny

Thierry Priol

To my parents Vasilios and Paraskevi ...

Résumé

Le traitement de problèmes rencontrés dans la construction de différentes familles d'applications a donné lieu à la définition et à la standardisation d'une couche logicielle qui se situe entre l'application et le système d'exploitation sous-jacent. Cette couche est connue sous le nom de *middleware*, et fournit des solutions réutilisables aux problèmes récurrents dans la construction de logiciels complexes, comme l'hétérogénéité, l'interopérabilité, la sécurité, la tolérance aux fautes ou encore l'exécution de transactions. Un *middleware* est typiquement construit à partir de services fournis par une infrastructure. Des exemples connus de telles infrastructures sont celles conformes au standard Corba, Dcom ou encore Ejb. L'implémentation d'un logiciel s'appuyant sur des solutions réutilisables, fournies par les infrastructures *middleware*, simplifie de manière évidente le processus de développement du logiciel. Les développeurs de logiciels se trouvent affranchis de l'implémentation de protocoles de gestion des communications, de la sécurité ou encore de modèles transactionnels. Étant donnée une infrastructure *middleware*, le travail du développeur quant à la mise en œuvre d'un système d'exécution (ou *middleware*) pour une application donnée est la combinaison de services *middleware* disponibles de telle sorte que le système résultant satisfasse les exigences de l'application. L'effort à fournir porte donc sur la conception d'architectures *middleware* qui satisfont les exigences particulières des applications. Notre travail de thèse a porté sur l'exploitation de la notion d'architecture logicielle pour la construction systématique de *middleware*, adaptés aux applications. Plus précisément, nous proposons l'utilisation d'un langage de description d'architectures et d'un ensemble d'outils associés pour systématiser : la conception d'une architecture *middleware* qui réponde aux besoins d'une application donnée, l'intégration de cette architecture au sein de l'application, et la maintenance de cette architecture au regard des évolutions relatives aux exigences de l'application ou à la disponibilité des services *middleware*.

La solution proposée à la construction systématique de *middleware* adaptés aux besoins des applications comprend un système de stockage qui garde trace de l'historique des conceptions d'architectures *middleware*. Ce système de stockage contient en outre les implémentations d'architectures concrètes qui résultent des étapes de conceptions mémorisées. Nous introduisons par ailleurs une méthode pour la localisation systématique de chemins de conceptions, *i.e.* séquences d'étapes de conceptions, qui conduisent éventuellement à des architectures concrètes de *middleware*, satisfaisant les exigences d'une application donnée. Cette facilité est complétée par une méthode de développement d'implémentations d'architectures concrètes de *middleware*, qui peuvent être aisément intégrées au sein d'une application. Enfin, nous donnons une solution à l'adaptation dynamique d'un *middleware* construit suivant notre méthode, qui préserve l'exécution cohérente de l'application s'appuyant sur celui-ci. Une telle fonctionnalité est essentielle pour la maintenance d'une

architecture *middleware* au regard de l'évolution des exigences de l'application et de l'infrastructure *middleware*.

Abstract

The urgent need to deal with problems that are frequently met in many different families of application led to the evolution and standardization of a software layer that lies between the application and the underlying operating system. This layer is widely known as *middleware*, and provides reusable solutions to problems like heterogeneity, interoperability, security, transactions, fault tolerance etc. Middleware is typically build based on services provided by an infrastructure. Well known examples of such infrastructures are those complying to the CORBA standard, DCOM, EJB etc. Implementing software based on the reusable solutions provided by middleware infrastructures certainly reduces the cost of the overall software development process. Developers are released from implementing communication protocols, security policies, transaction models. Given a middleware infrastructure, all they have to do is combine available middleware services in a way that satisfies the requirements of a particular application. Hence, their effort is now focused on the design of a middleware architecture that satisfies the application requirements.

This thesis proposes using the software architecture paradigm for the systematic customization of middleware. More specifically, it proposes using an architecture description language, together with adequate tool support for designing a middleware architecture that meets requirements of an application, integrating this middleware architecture within the application, and maintaining this middleware architecture with respect to changes in the application requirements and the availability of middleware services.

The proposed support for the systematic customization of middleware comprises a middleware repository that is used to store a history of previous design steps. Moreover, the proposed middleware repository contains the implementations of the concrete middleware architectures that resulted from the aforementioned design steps. Furthermore, a systematic method is proposed for locating design paths might lead to concrete middleware architectures that satisfy the requirements of an application. The support for the systematic customization of middleware further includes a method for developing implementations of concrete middleware architectures that can be easily integrated within an application. Finally, the systematic customization of middleware proposes a foundation for adapting a middleware while preserving the consistent execution of the application that lies on top of it. Such an ability is crucial so as to correctly maintain a middleware architecture with respect to changes in the application requirements and the availability of middleware services.

Acknowledgments

First I would like to thank Professors Gul Agha, Jean-Pierre Banâtre, Gordon Blair and Dr. Thierry Priol for being members of my thesis committee and for their invaluable comments on my work and its perspectives.

I would further like to thank Michel Banâtre, Directeur de Recherche at Inria and the members of the group Solidor for their kind welcome, useful discussions and support.

Special thanks to my advisor Valérie Issarny for her support and patience during those three and a half years.

I must mention that I feel extremely grateful to meet Dr. Apostolos Kountouris, a real cynic, who helped me clear out my ideas regarding several philosophical and practical questions that puzzled me during my stay in France. Hasta Siempre !

Not to forget to mention the invaluable help of the following members of "The Cage"[1], Dr. Titos Saridakis, Dr. Nikolaos Paragios, Irini Fundulaki, Christos Kloukinas, and my best friend John Kortzhs. Due to a very strange coincidence I keep finding those people right beside me (for better, or worst :-) every-time I need them.

Getting to those who keep supporting my efforts and forgiving my mistakes from the moment that I was born, many thanks to my parents Paraskevi and Vasilios, and my sister Maria.

Least but not last I have to thank my mentors, Vasilios Zarras and Evangellos Kontosakkos, who are, fortunately, still not tired with giving me precious advises every-time I find myself into trouble.

[1] http://www.csd.uch.gr/~maraz/TheCage

Table des Matières

Avant-propos

La construction d'applications distribuées est à présent facilitée par l'émergence d'infrastructures *middleware* (*e.g.*, standard CORBA, EJB, DCOM) qui fournissent des solutions réutilisables aux problèmes récurrents dans la construction de systèmes logiciels distribués, comme l'hétérogénéité, l'interopérabilité, la sécurité, la tolérance aux fautes ou encore l'exécution de transactions. L'implémentation d'une application s'appuyant sur les solutions réutilisables fournies par une infrastructure *middleware*, simplifie de manière évidente le processus de développement du logiciel. Les développeurs de logiciels se trouvent affranchis de l'implémentation de protocoles de bas niveau relatifs à la gestion de la distribution. Étant donnée une infrastructure *middleware*, le travail du développeur quant à la mise en œuvre d'un système d'exécution distribuée (ou *middleware*) pour une application donnée se ramène à la combinaison de services *middleware* disponibles de manière à ce que le système résultant satisfasse les exigences de l'application. L'effort à fournir porte donc sur la conception d'architectures *middleware* qui satisfont les exigences particulières des applications.

Dans ce cadre, notre travail de thèse a porté sur la définition d'une méthode et d'outils associés pour la conception systématique d'architectures *middleware* répondant aux exigences des applications, à partir d'infrastructures *middleware*s données. Plus précisément, nous proposons l'utilisation d'un langage de description d'architectures et d'un ensemble d'outils associés pour systématiser :

- La conception d'une architecture *middleware* qui répond aux besoins d'une application donnée,
- L'intégration de cette architecture au sein de l'application, et
- La maintenance de cette architecture au regard des évolutions relatives aux exigences de l'application ou à la disponibilité des services *middleware*.

La solution proposée à la construction systématique de *middleware* adaptés aux besoins des applications comprend un système de stockage qui garde trace de l'historique des conceptions d'architectures *middleware*. Ce système de stockage contient en outre les implémentations concrètes d'architectures qui résultent des étapes de conceptions mémorisées. Nous introduisons par ailleurs une méthode pour la localisation systématique de chemins de conceptions, *i.e.*, séquences d'étapes de conceptions, qui conduisent éventuellement à des architectures concrètes de *middleware*, satisfaisant les exigences d'une application donnée. Cette facilité est complétée par une méthode de développement d'implémentations d'architectures concrètes de *middleware*, qui peuvent aisément être intégrées au sein d'une application. Enfin, nous donnons une solution à l'adaptation dynamique d'un *middleware*, construit suivant notre méthode, qui préserve l'exécution cohérente de l'application s'appuyant sur le *middleware*. Une telle fonctionnalité est essentielle pour la maintenance d'une architecture *middleware* au regard de l'évolution des exigences de l'application et de l'infrastructure *middleware*.

Ce document se décompose en deux parties :

- La première partie, rédigée en français, est un résumé de nos travaux de thèse.
- La seconde partie, rédigée en anglais, présente de manière détaillée les résultats de nos travaux de thèse.

Soulignons que la seconde partie constitue le cœur de ce document et peut-être lue individuellement. En revanche, la première partie ne peut être considérée comme reflétant l'ensemble de notre travail.

Solution à la construction systématique de middleware

Résumé étendu

I Introduction

Un *middleware* est une couche logicielle qui se situe entre le système d'exploitation et l'application. Il fournit des solutions réutilisables à des problèmes fréquemment rencontrés lors de la construction de différentes classes applications ; citons par exemple les problèmes relatifs à la gestion des communications, à la fiabilité ou encore à la sécurité des applications distribuées. Le *middleware* masque les détails d'implémentation ayant trait à la gestion de l'hétérogénéité et de l'interopérabilité au moyen d'un ensemble d'interfaces. En général, un *middleware* n'est pas un bloc monolithique, mais est construit à partir d'un ensemble de services *middleware*, composés de manière à satisfaire les exigences de l'application. Ainsi, développer un *middleware* pour une application donnée se décompose en les étapes suivantes :

(1) Concevoir une architecture concrète de *middleware*, qui satisfasse les exigences de l'application,

(2) Implémenter le *middleware* comme prescrit par l'architecture qui le décrit,

(3) Maintenir le *middleware* de manière à intégrer les éventuelles modifications apportées aux exigences de l'application ou à l'infrastructure *middleware* dont il dépend.

Dans ce document de thèse, nous proposons une méthode pour la construction systématique de *middleware* qui soient adaptés aux besoins des applications. La méthode proposée vise à simplifier les 3 étapes mentionnées ci-avant, lesquelles constituent le processus de développement de *middleware*. Dans ce chapitre, nous précisons plus avant la notion de *middleware*, motivons l'approche préconisée pour leur construction systématique, puis introduisons brièvement cette approche, laquelle est résumée dans l'ensemble des chapitres suivants de cette partie et détaillée dans la seconde partie de ce document.

I.1 Infrastructures middleware

Les principales caractéristiques associées au développement fructueux d'un logiciel sont semblables à celles retenues dans l'évaluation de toute activité de production. Le produit doit satisfaire les exigences du consommateur, et le compromis effectué entre la qualité du

produit et le temps nécessaire à sa réalisation doit être optimal. En général, une partie des exigences du consommateur est spécifique au produit considéré alors que le reste des exigences est commun à différents produits. Classifier les produits en fonction de leurs exigences permet d'identifier une hiérarchie de familles de produits. Dans le cas particulier des produits logiciels, les applications distribuées constituent une famille intéressante de produits du fait de la grande variété des exigences communes à ces applications mais aussi du fait de la difficulté avérée de satisfaire ces exigences. Utiliser des solutions connues et largement adoptées par les concepteurs, pour satisfaire les exigences communes à différentes applications distribuées, permet d'économiser un temps précieux lors du développement d'une application donnée. Ce temps peut alors être mis à profit pour se concentrer pleinement sur le traitement des exigences spécifiques de l'application.

Les considérations indiquées ci-avant ont conduit à l'évolution de la notion de *middleware*. Un *middleware* est une couche logicielle qui se situe entre le système d'exploitation et l'application [9], et qui fournit des solutions connues et réutilisables, à des problèmes fréquemment abordés lors de la construction d'applications distribuées (*e.g.*, hétérogénéité, interopérabilité, sûreté de fonctionnement). La plupart des infrastructures *middleware* disponibles à ce jour sont fondées sur le paradigme *orienté-objet*. De ce fait, ces infrastructures imposent en général un *modèle objet* qui doit être utilisé pour la construction d'applications et de *middleware*, à partir des services fournis par l'infrastructure. Dans ce cadre, les *objets* sont les principales entités qui constituent une application ou un *middleware*. Tout objet exhibe alors une interface qui définit les opérations pouvant être appelées pour accéder et manipuler l'état interne de l'objet. Une *interface* est généralement décrite au moyen d'un *langage de définition d'interfaces* (ou IDL pour *Interface Definition Language*), lequel est spécifique à l'infrastructure *middleware*. Les objets émettant des appels sont enfin qualifiés de *clients*, alors que les objets recevant des appels sont qualifiés de *serveurs*. Un appel d'opération peut ainsi être divisé en deux parties :

(1) La *requête*, de l'objet client vers l'objet serveur, pour le service offert *via* l'opération,

(2) La *réponse* de l'objet serveur à l'objet client, laquelle retourne le résultat de l'exécution de l'opération.

La réalisation d'un tel appel est effectuée au moins au moyen d'un *courtier –de requêtes–* (ou *broker*), lequel constitue le noyau de toute infrastructure *middleware*.

Précisément, un courtier offre les fonctionnalités minimales, nécessaires aux interactions entre objets, *i.e.*, il permet l'échange de requêtes et réponses entre objets [71], tout en gérant les problèmes d'interopérabilité et d'hétérogénéité, de manière transparente pour l'application. Les objets de l'application interagissent au moyen de *proxies*, lesquels composent les fonctionnalités offertes par le courtier. Un *proxy* se décompose en deux parties : le *proxy client* et le *proxy serveur*. Une interaction entre deux objets se déroule ainsi de la manière suivante, si l'on se concentre sur la requête associée à l'appel :

- Un objet client appelle une opération fournie par un *proxy* client.
- Le *proxy* client crée la requête correspondante.
- La requête est délivrée au *proxy* serveur.
- Le *proxy* serveur transforme la requête en un appel à l'opération fournie par l'objet serveur.

Le traitement de la réponse, après terminaison de l'exécution de l'opération par le serveur, s'effectue ensuite de manière duale.

Dans sa globalité, outre la provision d'un IDL et d'un courtier, une infrastructure *middleware* offre un ensemble d'outils pour simplifier la construction d'applications. Ces outils incluent au moins le moyen de générer des *proxies* à partir de la description IDL des interfaces fournies par les objets serveurs. De plus, une infrastructure *middleware* offre en général un ensemble de *services* réutilisables, qui sont à combiner avec le courtier afin d'adapter le *middleware* aux exigences particulières de l'application.

Depuis le début des années 1990, il y a eu différents efforts visant à définir des standards décrivant le comportement (souvent de manière informelle) et la structure, d'infrastructures *middleware* qui soient capables de supporter l'exécution d'une grande variété d'applications. Le modèle RM-ODP (*The Open Distributed Processing Reference Model*) constitue un tel standard [33]. Succinctement, les infrastructures *middleware* qui satisfont ce standard doivent fournir le moyen de construire des applications suivant le modèle objet RM-ODP. Des standards comme RM-ODP restent abstraits et ne donnent pas de détails quant au comportement des fonctions de l'infrastructure. Toutefois, il existe des standards comme CORBA qui sont beaucoup plus précis pour ce qui est de ces aspects (*e.g.*, le service transactionnel de CORBA implante le protocole de validation à deux phases). La spécification de l'architecture CORBA (*The Common Object Request Broker Architecture*) [69] est parmi les tentatives les plus fructueuses, dans le domaine de la standardisation du comportement et de la structure d'une infrastructure *middleware*. Comme à l'accoutumé, le standard CORBA prescrit un modèle objet pour la construction des applications, lequel doit être supporté par toute infrastructure se voulant compatible avec le standard. Comme exemples connus d'infrastructures conformes au standard CORBA, indiquons notamment ORBIX [32], MICO [74] et OMNIORB2 [47]. En général, ces infrastructures fournissent en outre des fonctionnalités supplémentaires qui ne font pas partie du standard. Hormis les infrastructures *middleware* standard, il en existe d'autres qui sont développées, soit par des sociétés réputées comme Sun ou Microsoft, soit par des instituts de recherche. Par exemple, Sun a développé l'infrastructure EJB (*Enterprise Java Beans*) [85] ; Microsoft a mis en place l'infrastructure basée sur le modèle DCOM (*Distributed Object Component Model*) [58].

Quelque soit l'infrastructure *middleware* considérée, l'utilisation d'une telle infrastructure pour la construction d'une application distribuée permet un gain de temps non négligeable pour le développeur, tout en lui permettant d'aboutir à une architecture d'applications structurée. Toutefois, la conception d'un *middleware* qui satisfait les exigences d'une

application donnée requiert également un investissement significatif. Hors, les infrastructures *middleware* existantes n'offrent aucune aide quant à la simplification de cette tâche.

I.2 Motivation

Considérant notre présentation de la section précédente, le développement de systèmes logiciels distribués est simplifié par l'existence d'infrastructures *middleware*, qui :

- Traitent des problèmes récurrents d'hétérogénéité et d'interopérabilité, de manière transparente pour les applications, et
- Fournissent des services réutilisables résolvant des problèmes de gestion de la distribution, souvent rencontrés dans la pratique.

De manière plus précise, le développement de systèmes logiciels distribués se trouve décharger de l'implémentation de protocoles pour la gestion de l'interopération, de la sécurité des interactions entre les entités de l'application, de la tolérance aux fautes ou encore de l'exécution atomique et isolée d'actions concurrentes. Néanmoins, cette simplification avérée ne doit pas amener à considérer que la construction de *middleware* pour les applications devient une tâche triviale. Si l'on identifie un certains nombre d'avantages à la construction d'une application distribuée à partir d'une infrastructure *middleware*, le processus de développement de l'application n'en intègre pas moins la conception de l'architecture *middleware* satisfaisant les exigences de l'application. En général, la conception d'une architecture *middleware* relève du travail conjoint d'un architecte et d'un concepteur de systèmes logiciels.

À partir d'une description abstraite des exigences du consommateur pour un système logiciel donné, l'architecte a en charge de raffiner progressivement ces exigences pour notamment révéler une ou plusieurs *architectures concrètes de middleware* (*i.e.*, des architectures logicielles pouvant donner lieu à une implantation), qui satisfont les exigences initiales [64]. Raffiner correctement des exigences abstraites en des architectures logicielles concrètes est une tâche complexe. Même dans des cas simples comme l'étude de cas présenté dans la référence [64], le processus de raffinement requiert un investissement non négligeable de la part de l'architecte.

Le concepteur de logiciels utilise pour sa part le résultat du travail de l'architecte. Il a alors à sa disposition une architecture logicielle (concrète), *i.e*, une configuration de composants fournissant des propriétés données mais n'incluant aucune indication quant à son implantation. Une des principales responsabilités du concepteur est ensuite de vérifier la disponibilité de composants logiciels réutilisables dont les propriétés, lorsqu'elles sont combinées comme prescrit par l'architecture donnée, satisfont les exigences initiales, spécifiées pour le logiciel considéré. Cette tâche requiert également un investissement significatif, notamment si l'on considère les solutions *middleware* réutilisables qui sont disponibles ou en

cours d'élaboration à ce jour. La conception de *middleware* s'avère encore plus compliquée si les exigences initiales de l'application (du point de vue de la gestion de la distribution) intègrent des propriétés, éventuellement conflictuelles. Considérons par exemple des exigences d'efficacité et de fiabilité. Le premier type de propriétés impose d'évaluer les différentes options de conception en terme de leur performance respective. Par exemple, un service qui offre la propriété d'atomicité au moyen d'un protocole de validation à deux phases est moins onéreux en terme de bande passante consommée, comparé à un service qui implante la même propriété au moyen d'un protocole de validation à trois phases. De la même manière, la réplication active pour assurer des propriétés de fiabilité est plus onéreuse en terme de bande passante consommée, comparée à la réplication passive qui offre des propriétés comparables. Elle est toutefois plus économique en terme de consommation de l'espace de stockage puisque la réplication passive requiert une journalisation des états du système. Il s'ensuit que la conception de tout architecture *middleware* doit pouvoir s'accompagner d'un prototypage rapide. En d'autres termes, il doit être possible d'assembler les services *middleware*, et d'intégrer le *middleware* résultant au sein de l'application, aussi vite que possible. Les exigences de fiabilité obligent, pour leur part, le concepteur à examiner les différentes options de réalisation en termes de leurs comportements exceptionnels, de leurs limites vis-à-vis des ressources mises à disposition, ou encore de leur dépendance envers le matériel. Enfin, les exigences de l'application, de même que la disponibilité des services *middleware* utilisés, peuvent évoluer tout au long de la durée de vie de l'application. De tels changements nécessitent la maintenance de tout *middleware* de manière à garantir leur prise en compte.

Le développement d'un système logiciel reste une tâche difficile, même lorsqu'il est construit à partir de solutions réutilisables, comme celles offertes par les infrastructures *middleware* pour ce qui concerne la gestion de la distribution. Ceci réduit considérablement le bénéfice avéré de la réutilisation de logiciel. Dans ce contexte, nous proposons une *méthode pour la construction systématique de* middleware *adaptés aux applications, tout en supportant leur évolution, pour répondre aux nouvelles exigences des applications ou encore intégrer les modifications des infrastructures* middleware *(ou* middleware customization *en anglais)*. Cette méthode vise à aider l'architecte, le concepteur et le développeur de systèmes logiciels distribués pour facilement concevoir, développer et maintenir un *middleware* destiné à une application donnée.

I.3 Construction systématique de middleware

Suite aux remarques de la section précédente, nous pouvons en déduire les définitions suivantes qui caractérisent l'ensemble du processus inhérent au développement de *middleware* pour les applications, étant données des infrastructures *middleware* :

- **Construire un middleware** se ramène à la conception d'une architecture *middleware* à partir d'un ensemble de services *middleware* disponibles, qui soit telle qu'elle

satisfait les exigences de l'application considérée.

- **Adapter un middleware** à une application relève de l'intégration de l'implémentation des différents éléments constituant une architecture *middleware* concrète, en un *middleware* puis à composer ce résultat avec l'application.

- **Faire évoluer un middleware** se ramène enfin à une adaptation dynamique du *middleware* de manière à refléter les modifications apportées aux exigences de l'application ou encore à l'infrastructure *middleware*

Nous introduisons ci-après nos solutions à ces trois composantes du processus de développement de *middleware*. Le processus de conception d'une architecture *middleware* (*i.e.*, *Construction de middleware*) peut bénéficier de la réutilisation d'architectures *middleware* disponibles, qui satisfont les exigences de l'application considérée. À partir de cette remarque, le premier objectif de notre travail a été de concevoir un système de stockage des architectures *middleware* disponibles. Ce système peut être systématiquement examiné par un architecte pour *rechercher* une architecture concrète qui raffine des exigences abstraites spécifiées par un client, du point de vue de la gestion de la distribution. Le système de stockage doit être en outre conçu de manière à simplifier le travail du concepteur lorsqu'il essaie de trouver des services *middleware* dont les propriétés permettent de satisfaire les exigences abstraites initiales, ces services étant composés suivant l'architecture concrète qui lui est fournie par l'architecte. Nous proposons également une méthode pour l'intégration systématique d'un *middleware* au sein d'une architecture de système logiciel donnée (*i.e.*, *Adaptation du middleware à l'application*). Cette méthode, qui est indépendante de toute infrastructure *middleware*, lorsqu'elle est combinée avec un outil de génération de code associé, simplifie la tâche du développeur. Elle permet de plus un prototypage rapide, ce qui rend moins onéreux l'évaluation des performances des différentes options possible pour l'adaptation d'une architecture *middleware* particulière. Par la suite, nous employons le terme *synthèse de middleware* pour désigner les processus de recherche et d'intégration de *middleware*, susmentionnés. Faire évoluer un *middleware* lorsque ceci peut être dû à la fois à une modification des exigences de l'application et à celle de l'infrastructure *middleware* peut entraîner des modifications structurelles importantes au niveau de l'architecture *middleware*. Par conséquent, dans le pire des cas, faire évoluer un *middleware* requiert de l'échanger avec un nouveau *middleware*. Ceci doit alors être réalisé en un temps fini, suivant une approche qui introduit des perturbations minimales quant à l'exécution de l'application et qui préserve la cohérence de l'application. Un dernier objectif de notre travail a par conséquent été de proposer une conception d'architecture concrète de middleware qui puisse être échangée, tout en satisfaisant les contraintes susmentionnées.

Les chapitres suivants de cette première partie de notre document de thèse, proposent un résumé de l'ensemble de notre solution à l'aide au développement de *middleware*. Le chapitre II introduit un langage de description d'architectures logicielles ; ce langage est exploité tant pour la description des architectures des applications que pour celle des architectures *middleware* et constitue l'élément sous-tendant notre solution. Les chapitres III

et IV précisent ensuite respectivement les méthodes et outils associés à la synthèse puis à l'évolution (ou maintenance) de *middleware*. Enfin, un résumé de notre contribution et de nos perspectives suite à notre travail est présenté dans le chapitre V. Rappelons, que les résultats de notre travail sont plus amplement détaillés dans la seconde partie du document, laquelle comprend notamment une comparaison de nos solutions au regard des travaux apparentés ainsi qu'une étude de cas dont le développement progressif illustre l'ensemble des éléments de notre proposition.

II Description d'architectures

Dans la pratique, tant le comportement que la structure d'un *middleware* sont spécifiés de manière semi-formelle. Précisément, un *middleware* est accompagné de :

- Une description d'interfaces, donnée au moyen d'un IDL, spécifique à l'infrastructure dont le *middleware* dépend, et de
- Une description de ses propriétés structurelles et comportementales, laquelle est donnée au moyen d'expressions d'un langage naturel.

La construction systématique de logiciels requiert toutefois une description précise et non ambiguë des propriétés et de la structure du logiciel. Afin de supporter une telle description, la communauté travaillant dans le domaine des architectures logicielles, sous-domaine du génie logiciel, a proposé des langages de description d'architectures (ou ADL pour *Architecture Description Language*) [17, 35, 56], *i.e.*, des langages qui fournissent le moyen de décrire formellement la structure et les propriétés qui caractérisent le comportement d'un logiciel complexe.

Ce chapitre introduit un ADL pour la *configuration systématique de middleware*[1]. L'ADL proposé fournit le moyen de décrire les composants constituant un système logiciel distribué et les connecteurs utilisés pour la réalisation des interactions entre les composants. L'ADL est en outre combiné avec un modèle de spécification formelle des propriétés caractérisant le comportement d'un logiciel. Ce modèle s'appuie sur la logique temporelle linéaire, mais pourrait tout aussi bien s'appuyer sur d'autres formalismes existants.

II.1 ADL pour la configuration systématique de middleware

À partir de la notion d'architecture logicielle et dans le but de configurer systématiquement des *middleware*, nous proposons un ADL qui permet de caractériser la structure et les pro-

[1] Le terme *configuration systématique* de *middleware* est ici employé, tout comme dans la suite de cette première partie, pour signifier la construction de *middleware* évolutifs, adaptés aux exigences des applications, *i.e.*, comme une traduction abrégée du terme anglais *customized middleware*.

priétés d'un *middleware*. L'ADL permet ainsi de décrire la structure et les propriétés de *composants* logiciels, lesquels peuvent être des composants de l'application ou des services *middleware*. L'ADL offre en outre le moyen de décrire des *connecteurs*, construits à partir de services *middleware*. La suite de cette section précise les définitions de ces éléments d'une architecture.

Définition des composants. La description d'un *composant* définit un *type*, lequel spécifie un ensemble d'*interfaces*, requises ou offertes par toute instance de ce composant. Le type composant est exploité pour créer de multiples *instances de composants*.

Afin de permettre la description de composants à différents niveaux de détail, le type composant permet également de déclarer une instance de *configuration*, laquelle réalise alors le composant. En d'autres termes, le composant correspond effectivement à un ensemble de composants interconnectés *via* un connecteur. Dans le cas où la description d'un composant définit une configuration, le composant est dit *composite*. Dans le cas contraire, le composant est dit *primitif*.

Une *interface* est du type particulier `interface`, lequel sert à décrire un ensemble d'opérations. La syntaxe utilisée pour décrire ces derniers types est similaire à celle utilisée par les langages IDL offerts par les infrastructures *middleware* disponibles à ce jour (*e.g.*, IDL de CORBA, constructeur d'interface du langage JAVA, langage MIDL de DCOM).

Enfin, le type composant permet de décrire les propriétés d'un composant. Ces propriétés décrivent le comportement observable du composant lorsqu'il interagit avec les autres éléments de l'architecture ; la spécification de propriétés est traitée plus avant dans la section suivante.

L'exemple présenté ci-dessous montre les notations utilisées dans ce document pour décrire les composants d'un système logiciel, qu'il s'agisse d'une application ou d'un *middleware*. Le composant donné est composite, constitué d'une configuration `hc` de type `HorizontalComposition`.

```
component Composite {
   requires // Liste des interfaces requises;
     InterfaceTypeA InterfaceInstanceA;
   provides // Liste des interfaces fournies;
     InterfaceTypeB InterfaceInstanceB;
   configuration HorizontalComposition hc;
   properties       // Comportement observable du composant };
interface InterfaceTypeA {
   // Liste des opérations requises;
   RetType op(in argType arg, out argType' arg', ...);};
interface InterfaceTypeB {
    ....; };
```

Définition des configurations. La description d'une configuration introduit un *type*, lequel définit un ensemble d'instances de composants et une instance de connecteur. Les instances de composants sont interconnectées en termes de liaisons entre leurs interfaces requises et fournies, les liaisons étant réalisées au moyen du connecteur. Le type configuration est introduit pour définir une ou plusieurs instances d'une configuration.

Nous donnons ci-après un exemple illustrant la syntaxe utilisée pour décrire une configuration. Cet exemple introduit un type configuration de nom `HorizontalComposition`, lequel définit deux instances de types `ConstituentC` et `ConstituentS`, interconnectés *via* le connecteur de type `MdwConn`.

```
configuration HorizontalComposition {
    instances // Liste des instances de composants et connecteur;
     ConstituentC C;
     ConstituentS S;
     MdwConn MdwConnInst;
    bindings // Liste des liaisons entre les interfaces fournies et requises;
     C.Interface to S.Interface through MdwConnInst;
   };
```

Définition des connecteurs. L'objectif de la configuration systématique de *middleware* est de concevoir et mettre en œuvre un connecteur qui représente une architecture *middleware*, laquelle est exploitée pour régir les interactions entre les composants de l'application de manière à satisfaire les exigences de cette dernière. Il s'ensuit qu'un connecteur définit les éléments architecturaux constituant l'architecture *middleware*. Ces éléments peuvent être des composants et des connecteurs plus primitifs, leurs instances étant composées *via* une configuration. Un connecteur a un *type* associé, lequel est utilisé pour définir différentes instances de connecteurs au sein d'une description ADL. Définir une instance de connecteur est équivalent à définir l'instance de la configuration correspondant au connecteur. Enfin, une description de connecteur comprend les propriétés fournies par celui-ci. Ces propriétés caractérisent les interactions entre les composants liés *via* une instance de ce connecteur. Par exemple, les propriétés d'un connecteur peuvent stipuler que les interactions entre les composants liés au moyen d'une instance de ce connecteur sont fiables et synchrones.

À partir de notre présentation de la section I.1, un connecteur mis en œuvre au moyen d'un *middleware* est construit à partir d'une *infrastructure middleware* existante, et représente une *architecture middleware*. Cette architecture définit au moins les composants correspondant aux *proxies* clients et serveurs, qui combinent les fonctionnalités fournies par un courtier de manière à permettre aux composants de l'application d'interagir. De plus, un connecteur définit un type de configuration qui comprend des instances de composants correspondant aux proxies clients et serveurs. Les liaisons entre les proxies client et serveur sont réalisées au moyen d'un connecteur plus primitif (*e.g.*, un connecteur primitif RPC ou encore un connecteur primitif TCP/IP). À l'exception des proxies client et serveur,

un connecteur peut s'appuyer sur des composants représentant des services *middleware*, autres que le courtier, fournis par l'infrastructure *middleware*. La configuration d'un connecteur peut ainsi comprendre des instances de composants correspondant à des services *middleware*.

Étant donné que le principal objectif de la configuration systématique de *middleware* est de concevoir et mettre en œuvre un connecteur satisfaisant les exigences de l'application, la description ADL d'un connecteur n'a pas à détailler les éléments et la structure des connecteurs. Il s'ensuit que la plupart des caractéristiques d'un connecteur décrites ci-avant peuvent être omises lors de la description de la structure et des exigences d'une application. Plus précisément, la seule caractéristique d'un connecteur qui nous intéresse ici est la propriété fournie par un connecteur, comme illustré par l'exemple suivant.

```
connector MdwConn {
   properties
      // e.g Communication fiable
};
```

II.2 Spécification des propriétés d'un middleware

Comme indiqué au début de ce chapitre, la spécification d'un *middleware* est en général semi-formelle, en ce sens que la structure et les propriétés du *middleware* sont définies au moyen d'expressions données dans un langage naturel. L'ADL que nous avons introduit jusqu'ici permet de décrire formellement la structure d'un *middleware*. Comme nous le voyons ci-après, les propriétés d'un *middleware* sont spécifiées quant à elle au moyen de la logique temporelle. Nous utilisons les notations classiques de cette logique pour la définition des propriétés [53], ces notations étant en outre rappelées dans le deuxième chapitre de la seconde partie de ce document.

Les propriétés caractérisant un composant ou un connecteur sont données comme des axiomes composant une théorie. La syntaxe utilisée dans les définitions données ci-après est basée sur la syntaxe du démonstrateur de théorèmes STEP (*Stanford TEmporal logic theorem Prover*) [12] ; STEP supporte les types primitifs comme les entiers (dénotés par *int*), les booléens (dénotés par *bool*), les réels (dénotés par *real*), et permet la définition de types complexes comme les tableaux (définis par `array` [*type*] `of` *type*), les n-uplets (définis par $(type, type', \ldots, type'')$), les structures (définies par $\{id : type, id' : type', \ldots id'' : type''\}$) *etc.*. De plus, $X[i]$ dénote le $i^{ème}$ élément du tableau X; $\sharp(i)t$ dénote le $i^{ème}$ élément du n-uplet t; $t.id$ dénote l'élément id de la structure t.

Afin de décrire en logique temporelle, les propriétés d'un *middleware* requises par une application ou fournies par un *middleware*, nous devons définir un modèle de spécification formelle de base, qui consiste en :

- Le vocabulaire des types de base, utilisés pour décrire les éléments architecturaux et les caractéristiques architecturales qui définissent récursivement ces éléments, *e.g.*, composants, configurations, interfaces, requêtes, opérations, défaillances, *etc.*.
- Les axiomes qui décrivent l'utilisation correcte des éléments du vocabulaire.

Suivant la référence [64], un tel modèle de spécification formelle, combiné avec un ensemble d'axiomes donnant les propriétés offertes par un ensemble de connecteurs, définit un *style architectural*. Toujours en s'appuyant sur la terminologie de cette même référence, le modèle de spécification utilisé pour décrire les propriétés d'un *middleware* est appelé ci-après une *théorie de style architectural* (*pour la spécification des propriétés d'un middleware*). D'un point de vue pratique, cette théorie est définie comme une théorie STeP, nommée `Base-Architectural-Style`, dont les éléments de base sont donnés ci-après.

- `type` $\mathcal{R}$, utilisé dans les spécifications pour définir les éléments de type requête. Une requête est ainsi définie dans une spécification comme une valeur de type $\mathcal{R}$:

$$value\ req : \mathcal{R}$$

- `type` $\mathcal{UID}$, utilisé dans les spécifications pour définir des identificateurs uniques afin de distinguer les valeurs identiques. Par exemple, les requêtes sont typiquement non uniques, et les identificateurs sont introduits pour les distinguer.
- `type` $\mathcal{C}$, utilisé pour définir les éléments de type composant, *i.e.*, les instances de composants. Plus précisément, le type composant est défini comme étant une structure à trois éléments :

$$\texttt{type}\ \ \mathcal{C} = \{I_p, I_r, \Sigma_C\},$$

où $I_p = \{i : \mathcal{I}_p\}$ est un ensemble d'interfaces fournies, $I_r = \{i : \mathcal{I}_r\}$ est un ensemble d'interfaces requises, et $\Sigma_C = \{\sigma : \Sigma\}$ est un ensemble d'états possibles. Ainsi, une instance de composant C dans la spécification d'une propriété est une valeur de type $\mathcal{C}$:

$$value\ C : \mathcal{C}$$

- `type` Σ, utilisé dans les spécifications pour caractériser des états de composants :

$$\sigma : \Sigma$$

- `type` $\mathcal{I}_r$, utilisé dans les spécifications pour définir des instances d'interfaces requises. Plus précisément, $\mathcal{I}_r$ est défini comme étant un ensemble d'opérations requises :

$$\texttt{type}\ \ \mathcal{I}_r = \{o_{C_i} : \mathcal{O}_r\},$$

- `type` $\mathcal{O}_r$, utilisé dans les spécifications pour définir des opérations requises. Plus précisément, le type des opérations requises est défini comme une fonction qui prend en entrée l'état du composant appelant et un identificateur unique, et qui retourne une requête correspondant à l'appel :

$$\texttt{type}\ \ \mathcal{O}_r : \Sigma * \mathcal{UID} \rightarrow \mathcal{R}$$

  Ainsi, une opération requise est une instance du type donné ci-dessus, *i.e.*, $value\ o : \mathcal{O}_r$, et $rang(o)$ dénote l'ensemble des requêtes que o retourne.

  De plus, l'axiome suivant est supposé vérifié pour les opérations d'une interface requise :

  `AXIOM Require-Unique-Signatures`
  $\forall I : \mathcal{I}_r,$
  $o_i, o_j : \mathcal{O}_r \mid$
  $((o_i \in I) \wedge (o_j \in I) \wedge (o_i \neq o_j)) \Rightarrow rang(o_i) \cap rang(o_j) = \emptyset$

- `type` $\mathcal{I}_p$, utilisé dans les spécifications pour définir les instances d'interfaces fournies. Plus précisément, $\mathcal{I}_p$ est défini comme étant l'ensemble des opérations fournies :

$$\texttt{type}\ \ \mathcal{I}_p = \{o_{C_i} : \mathcal{O}_p\},$$

- `type` $\mathcal{O}_p$, utilisé dans les spécifications pour définir les opérations fournies. Plus précisément, le type d'une opération fournie est défini comme une fonction prenant en paramètres l'état du composant appelé, une requête, et un identificateur unique, et qui retourne en résultat l'état du composant appelé après traitement de la requête :

$$\texttt{type}\ \ \mathcal{O}_p : \Sigma * \mathcal{R} * \mathcal{UID} \rightarrow \Sigma$$

  Ainsi, une opération fournie o est une instance de ce type, *i.e.*, $valueo : \mathcal{O}_p$, et $dom(o)$ dénote l'ensemble des n-uplets que o accepte en paramètres.

  Enfin, l'axiome suivant est supposé vérifié pour les opérations d'une interface fournie :

  `AXIOM Provide-Unique-Signatures`
  $\forall I : \mathcal{I}_p,$
  $o_i, o_j : \mathcal{O}_p \mid$
  $((o_i \in I) \wedge (o_j \in I) \wedge (o_i \neq o_j)) \Rightarrow dom(o_i) \cap dom(o_j) = \emptyset$

- Le symbole $\rightsquigarrow$ est utilisé dans une spécification pour dénoter une fonction booléenne qui prend en paramètres deux instances d'interfaces, et qui retourne *vrai* si ces instances sont interconnectées, et *faux* sinon :

$$\rightsquigarrow : \mathcal{I}_r * \mathcal{I}_p \rightarrow \{true, false\}$$

De plus, l'axiome suivant est vérifié :

AXIOM Type-Checking
$\forall i : \mathcal{I}_r,$
$j : \mathcal{I}_p \mid$
$(i \rightsquigarrow j) \Rightarrow$
$(\forall o_r : \mathcal{O}_r \mid$
$(o_r \in i) \wedge$
$(\exists o_p : \mathcal{O}_p \mid (o_p \in j) \wedge (rang(o_r) = \sharp(2) dom(o_p)))$

- $\mathcal{CONF}$ est utilisé dans les spécifications pour dénoter une configuration. Plus précisément, le type configuration est défini comme un ensemble d'instances de composants liées entre elles :

$$\texttt{type}\ \ \mathcal{CONF} = \{C : \mathcal{C} \mid (\exists C' : \mathcal{C} \mid (C' \in \mathcal{CONF}) \wedge (\exists i, i' \mid (i \in \sharp(I_r)C \vee i \in \sharp(I_p)C) \wedge (i' \in \sharp(I_r)C' \vee i \in \sharp(I_p)C') \wedge ((i \rightsquigarrow i') \vee (i' \rightsquigarrow i))))\}$$

- Le symbole [ ] est utilisé dans une spécification pour dénoter une fonction non interprétée qui prend en arguments un ensemble d'instances de composants et un état, et qui retourne *vrai* si l'ensemble de composants est dans cet état à l'instant considéré :

$$[\,] : \{C : \mathcal{C}\} * \Sigma \rightarrow \{true, false\}$$

- *response*() de type $\mathcal{O}_r$ est une instance d'une opération requise qui prend en arguments un identificateur unique $rid : \mathcal{UID}$ et l'état du composant appelé $C : \mathcal{C}$, et retourne une réponse $resp : \mathcal{R}$ à la requête associée à rid.

- *response*() de type $\mathcal{O}_p$ est une instance d'opération fournie qui prend en entrée l'état d'un composant appelant $C : \mathcal{C}$, une réponse $resp : \mathcal{R}$ et l'identificateur unique $rid : \mathcal{UID}$ qui lui est associé, et qui retourne l'état du composant appelant après qu'il ait reçu la réponse.

- Le symbole $failure()$ est utilisé dans une spécification pour dénoter une fonction qui prend en argument une requête et retourne *vrai* si la requête indique une exception système, et *faux* sinon :

$$failure : \mathcal{R} \rightarrow \{true, false\}$$

À partir des définitions introduites ci-avant, il est possible d'introduire les définitions suivantes (ou *macros*), qui sont utilisées dans les spécifications pour caractériser les interactions entre les composants d'une configuration.

- La définition suivante est vérifiée pour deux instances de composants, C, C', si ces instances sont des éléments d'une configuration $Conf$, et la requête req est le résultat de l'opération o qui appartient à l'interface ir requise par C, laquelle est liée à l'interface ip fournie par C' :

$$\begin{array}{l}
\texttt{macro} \\
\quad Call(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad \exists o : \mathcal{O}_r, \\
\qquad ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad ((ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in ir) \wedge \\
\qquad\quad (req \in rang(o)) \wedge \ominus[C, \sigma] \wedge req = o(\sigma, rid))
\end{array}$$

- La définition suivante est vérifiée pour deux instances de composants, C, C', si ces instances sont des éléments d'une configuration $Conf$, et C' est dans un état qui résulte de l'appel à l'opération o de l'interface ip fournie par C', laquelle est liée à l'interface ir requise par C :

$$\begin{array}{l}
\texttt{macro} \\
\quad UpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad (\exists ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad o : \mathcal{O}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad (ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in \sharp(I_p)C') \wedge \\
\qquad\quad ((\sigma, req, rid) \in dom(o)) \wedge \\
\qquad\quad \Diamond Call(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge [C', o(\sigma, req, rid)]
\end{array}$$

- La définition suivante est vérifiée pour deux instances de composants, C, C', si ces instances sont des éléments de la configuration $Conf$, et C' retourne une réponse $resp$ à la requête req du composant C :

$$\begin{array}{l}\texttt{macro}\\ \quad ReturnUpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) =\\ \qquad \exists req : \mathcal{R},\\ \qquad \sigma : \Sigma \mid\\ \qquad\quad \Diamond UpCall(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge resp = response(\sigma, rid)\end{array}$$

- La définition suivante est vérifiée pour deux instances de composants, C, C', si ces instances sont des éléments d'une configuration $Conf$ et C est dans un état qui résulte de la réception de la réponse $resp$ de la requête req émise à C' :

$$\begin{array}{l}\texttt{macro}\\ \quad ReturnCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) =\\ \qquad \exists \sigma : \Sigma \mid\\ \qquad\quad \Diamond ReturnUpCall(Conf, C, C', resp, rid) \wedge\\ \qquad\qquad \ominus[C, \sigma] \wedge [C, response(\sigma, resp, rid)]\end{array}$$

À partir de la théorie de style architectural `Base-Architectural-Style` dont les éléments ont été introduits ci-avant, il est possible de décrire différentes propriétés élémentaires d'un *middleware* (*e.g.*, communication Rpc fiable). De plus, avec des extensions simples du style donné, nous pouvons également spécifier des propriétés plus complexes (*e.g.*, communication Rpc transactionnelle). L'utilisation du modèle de spécification introduit pour caractériser différentes propriétés de *middleware* est abordée plus avant dans la seconde partie de ce document, notamment au travers d'un exemple détaillé qui illustre l'ensemble de notre proposition.

III Synthèse de middleware

Le processus de synthèse systématique de *middleware* comprend :

(1) La conception d'une architecture concrète de *middleware* qui raffine les exigences abstraites d'une application donnée (*i.e.*, construction du *middleware* suivant nos définitions de la section I.3) ;

(2) La mise en œuvre de l'architecture concrète identifiée, à partir de services *middleware* réutilisables(*i.e.*, adaptation du *middleware* suivant nos définitions de la section I.3).

Ce chapitre présente les principes de notre solution à la simplification des étapes susmentionnées. Dans un premier temps, afin de donner une idée précise de ce que recouvre la synthèse de *middleware*, nous définissons plus avant la notion d'architecture *middleware*. Nous introduisons également la relation de raffinement utilisée pour la synthèse systématique de *middleware* à partir de la spécification des exigences abstraites (ou propriétés attendues pour le *middleware*) de l'application. Notre solution s'appuie notamment sur les travaux dans le domaine du raffinement d'architectures logicielles. Cette relation est exploitée pour la structuration d'un système de stockage de *middleware*, ainsi que pour la recherche systématique d'architectures *middleware* qui raffinent des exigences abstraites d'applications. Enfin, nous abordons l'assemblage et l'intégration d'une implémentation de *middleware*, au sein d'une application, étant données la description architecturale de cette dernière et l'architecture concrète de *middleware* satisfaisant les exigences de l'application.

III.1 Architecture middleware

Un des objectifs de notre travail est de mécaniser la conception et l'implantation d'une architecture concrète de *middleware*, qui permet aux composants de l'application d'interagir de manière à satisfaire leurs exigences vis-à-vis du système sous-jacent. Une description brève des caractéristiques de base d'une architecture *middleware* a déjà été donnée dans la section II.1. Néanmoins, avant de détailler le processus de synthèse de *middleware* que nous proposons, il apparaît nécessaire de préciser la notion d'architecture concrète de *middleware* ainsi que la signification du raffinement des exigences abstraites d'une application en une architecture concrète de *middleware*.

Architecture concrète de *middleware*. Comme indiqué dans la section II.1, une architecture concrète de *middleware* comprend des composants dont les implémentations correspondent aux *proxies* clients et serveurs. Ces *proxies* combinent les fonctionnalités fournies par un courtier de manière à permettre aux composants de l'application d'interagir suivant leurs exigences vis-à-vis des propriétés du système d'exécution sous-jacent. Les liaisons entre les *proxies* clients et serveurs sont réalisées au moyen de connecteurs plus primitifs comme, par exemple, un connecteur TCP/IP ou encore un connecteur RPC.

À l'exception des *proxies*, une architecture concrète de *middleware* peut comprendre des composants qui correspondent aux services fournis par l'infrastructure *middleware* sous-jacente (*e.g.*, sécurité, transactions, etc). Les services *middleware* peuvent être utilisés explicitement par l'application, signifiant qu'il existe des *proxies* clients et serveurs qui permettent aux composants de l'application d'interagir avec le service. De la même manière, les services *middleware* peuvent être utilisés explicitement par d'autres services *middleware*. Dans ce cas, un service joue le rôle d'un client, requérant une interface fournie par l'autre service, lequel joue le rôle du serveur. Ici aussi, cette interaction est réalisée au moyen de *proxies* client et serveur.

Les fonctionnalités fournies par les services *middleware* peuvent également être combinées avec celles du courtier au sein d'un *proxy*. Ceci permet aux composants de l'application d'interagir de façon à satisfaire les exigences de l'application, de manière transparente pour cette dernière. Par exemple, il existe des infrastructures *middleware* comme ORBIX et MICO, qui permettent d'exécuter des opérations au sein des *proxies* clients, juste avant ou juste après l'émission d'une requête au serveur. Néanmoins, si une telle flexibilité n'est pas offerte par l'infrastructure *middleware* utilisée, une solution à la combinaison des fonctionnalités de services *middleware* avec celles du courtier consiste à utiliser des composants qui réalisent des *proxies* complexes. De tels composants sont connus sous le nom de *wrappers*. En général, l'utilisation de *wrappers* permet une implémentation structurée de *proxies* clients et serveurs complexes. Mais, cela rend plus difficile la tâche des développeurs.

D'un point de vue architectural, un *wrapper* est semblable à un *proxy* : un *wrapper* client est utilisé par un composant client pour émettre des requêtes à un composant serveur. Plus précisément, le *wrapper* client transfert les requêtes émises par le client au *wrapper* serveur correspondant. De plus, le *wrapper* client implante des appels aux opérations fournies par les services *middleware* juste avant, ou juste après, le transfert des requêtes du composant client vers le *wrapper* serveur. Le *wrapper* serveur délivre ensuite les requêtes des clients au *proxy* client du composant serveur. De plus, le *wrapper* serveur peut appeler des opérations de services *middleware* juste avant, ou juste après, le transfert d'une requête au *proxy* client du composant serveur.

À titre d'illustration, la figure III.1 donne un exemple d'architecture concrète de *middleware* qui régit les interaction entre des composants d'une application donnée.

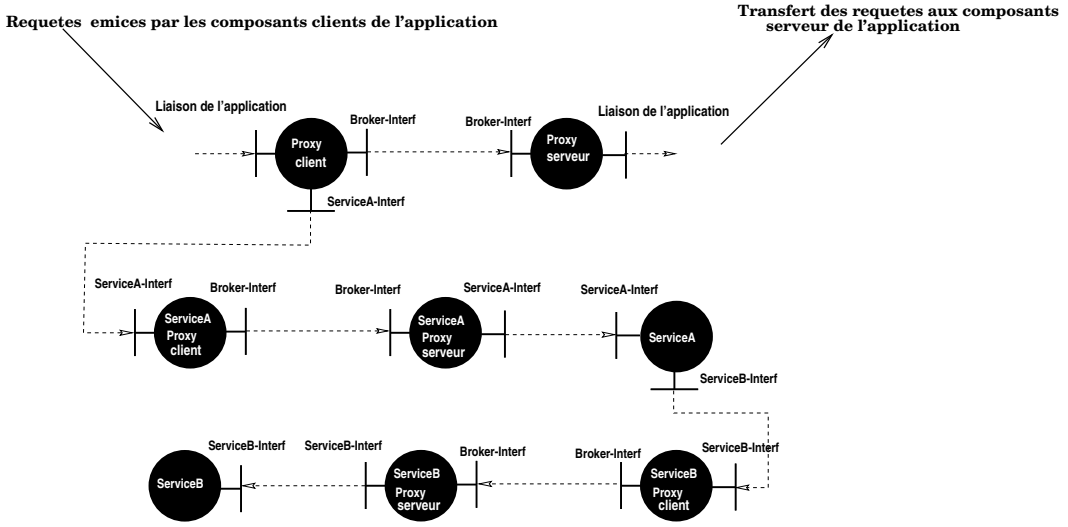

Figure III.1 Une architecture concrète de *middleware*

Raffinement d'architecture *middleware***.** Le raffinement d'une architecture en une architecture plus concrète signifie préciser la structure et les propriétés des éléments constituant l'architecture. Ainsi, pendant une étape de raffinement, l'architecte peut décomposer un élément de l'architecture abstraite en une configuration d'éléments architecturaux plus concrets. Pour vérifier la correction de cette étape, l'architecte doit alors prouver que la configuration d'éléments concrets fournit les propriétés de l'élément abstrait. Les propriétés offertes par une configuration sont la combinaison des propriétés fournies par les éléments concrets la constituant. Cette combinaison peut être dérivée à partir de la manière dont les composants sont interconnectés. Dans le contexte de ce document, nous supposons que les propriétés fournies par une architecture *middleware* sont décrites à partir du style architectural de base, `Base-Architectural-Style`, donné dans la section II.2, ou à partir d'extensions de ce style. Il s'ensuit qu'étant données deux théories, Θ et Θ', qui décrivent les propriétés respectivement fournies par les architectures *middleware*, M et M', nous obtenons que l'architecture *middleware* M raffine correctement l'architecture M', ce que nous notons $M \underset{refines}{\Rightarrow} M'$, si et seulement si la condition suivante est vérifiée :

$$M \underset{refines}{\Rightarrow} M' \equiv \forall P' \in \Theta' \mid \exists P \in \Theta \mid P \Rightarrow I(P')$$

Cette condition du raffinement correct d'architectures est inspirée des travaux présentés dans la référence [59] qui présente un moyen d'identifier sous quelles conditions une fraction de logiciel en raffine une autre. L'objectif de cette proposition est de structurer un système de stockage de modules logiciels de manière à permettre leur recherche efficace.

Étant donnés les éléments introduits dans cette section, relatifs à la définition des architectures *middleware* et à leur raffinement, nous sommes en mesure de présenter notre solution à la synthèse systématique de *middleware*.

III.2 Recherche de middleware

Étant donnée la relation de raffinement introduite dans la section précédente, et afin de simplifier la tâche de l'architecte et du concepteur d'applications lorsqu'ils tentent de raffiner les exigences abstraites d'une application en une architecture concrète de *middleware* les satisfaisant, la configuration systématique de *middleware* s'appuie sur un système de stockage structuré des architectures *middleware*.

Le système de stockage proposé peut être systématiquement examiné par l'architecte pour retrouver une architecture concrète de *middleware* qui raffine les exigences d'une application donnée. De plus, le système de stockage peut être utilisé par le concepteur pour retrouver des implémentations d'architectures concrètes de *middleware*, ou des implémentations d'éléments architecturaux qui peuvent être utilisés pour assembler l'implémentation d'une architecture concrète de *middleware*. Cette section détaille la structure du système de stockage de *middleware* proposé, puis présente chacune des étapes relatives à la recherche systématique d'un *middleware*.

III.2.1 Système de stockage de middleware

Afin d'être utile à l'architecte et au concepteur d'applications, le système de stockage de *middleware* est organisé en deux parties :

- Un *système de stockage de conceptions de middleware* qui contient l'historique de toutes les étapes de raffinement réalisées par l'architecte, lors de la configuration de *middleware* destinés à des applications développées dans le passé.
- Un *système de stockage d'implémentations de middleware* qui contient les implémentations disponibles de :
  - composants *middleware* implantant des services fournis par une infrastructure existante, et des
  - architectures concrètes de middleware régissant les interaction entre les composants d'applications au regard de propriétés données.

Système de stockage de conceptions de *middleware*. À partir de la terminologie des systèmes de gestion de fichiers, le système de stockage de conceptions de *middleware* est une hiérarchie de répertoires. Chaque répertoire de la hiérarchie contient les informations relatives à une architecture gérant les interactions entre les composants d'applications. Plus précisément, cette information se décompose en deux fichiers :

- Le premier fichier contient une description de la structure de l'architecture middleware, spécifiée au moyen de l'ADL présenté dans la section II.1.

- Le second fichier contient une spécification formelle des propriétés fournies par l'architecture *middleware*. Cette spécification est donnée à partir du style architectural de base, présenté dans la section II.2, ou des extensions de ce style. Pratiquement, la spécification formelle des propriétés des architectures *middleware* est donnée sous la forme d'une théorie STeP qui inclut des définitions données dans la théorie `Base--Architectural-Style` ou dans des extensions de celle-ci.

Mis à part les deux fichiers indiqués, tout répertoire peut contenir des liens vers des implémentations disponibles de l'architecture *middleware* correspondante. Ces liens conduisent au système de stockage d'implémentations, lequel est décrit dans le paragraphe qui suit. Enfin, chaque répertoire peut contenir des liens vers des sous-répertoires, lesquels stockent des informations relatives à des architectures *middleware* qui raffinent celle décrite par le répertoire père. Il s'ensuit que les sous-répertoires reflètent des informations concernant des étapes de raffinement, précédemment réalisées par l'architecte. De manière générale, l'ensemble de la hiérarchie de répertoires reflète un historique des étapes de raffinement réalisées par l'architecte durant la conception de *middleware* pour des applications développées dans le passé. La racine de la hiérarchie contient les informations relatives à l'architecture *middleware* qui fournit une communication Rpc non fiable. La figure III.2 donne une vue simplifiée d'un système de stockage de *middleware*. En particulier, la partie gauche de la figure représente le système de stockage des conceptions.

Système de stockage d'implémentations de *middleware*. La structure du système de stockage d'implémentations de *middleware* est plus simple. Comme déjà indiqué, ce système contient des implémentations disponibles d'architectures et de composants *middleware*. Son rôle est d'aider le concepteur d'applications lorsqu'il essaie de retrouver une implémentation d'architecture concrète de *middleware* donnée par l'architecte, ou lorsqu'il recherche des implémentations de composants *middleware* pouvant être utilisés pour construire une architecture concrète de *middleware* non encore implémentée.

Par conséquent, le concepteur d'applications parcourt le système de stockage d'implémentations de *middleware* afin de trouver des éléments logiciels relativement spécifiques comme, par exemple, une architecture fondée sur le standard Corba ou encore un service Corba. De par cette caractéristique, le système de stockage d'implémentations de *middleware* est divisé en un certain nombre de sous-répertoires, chacun d'eux stockant des éléments logiciels d'une infrastructure *middleware* spécifique. Par exemple, nous pouvons typiquement imaginer un système contenant un répertoire Corba, un répertoire Dcom, et un répertoire Ejb. Le répertoire Corba inclut lui-même un certain nombre de sous-répertoires, lesquels sont spécifiques à différentes infrastructures conformes au standard (*e.g.*, Mico, Orbix, *etc.*).

Chaque répertoire correspondant à une infrastructure spécifique contient l'implémentation du courtier et les outils d'aide au développement au-dessus de cette infrastructure particulière (*e.g.*, compilateur Idl, *etc.*). De plus, ces répertoires sont divisés en deux sous-répertoires :

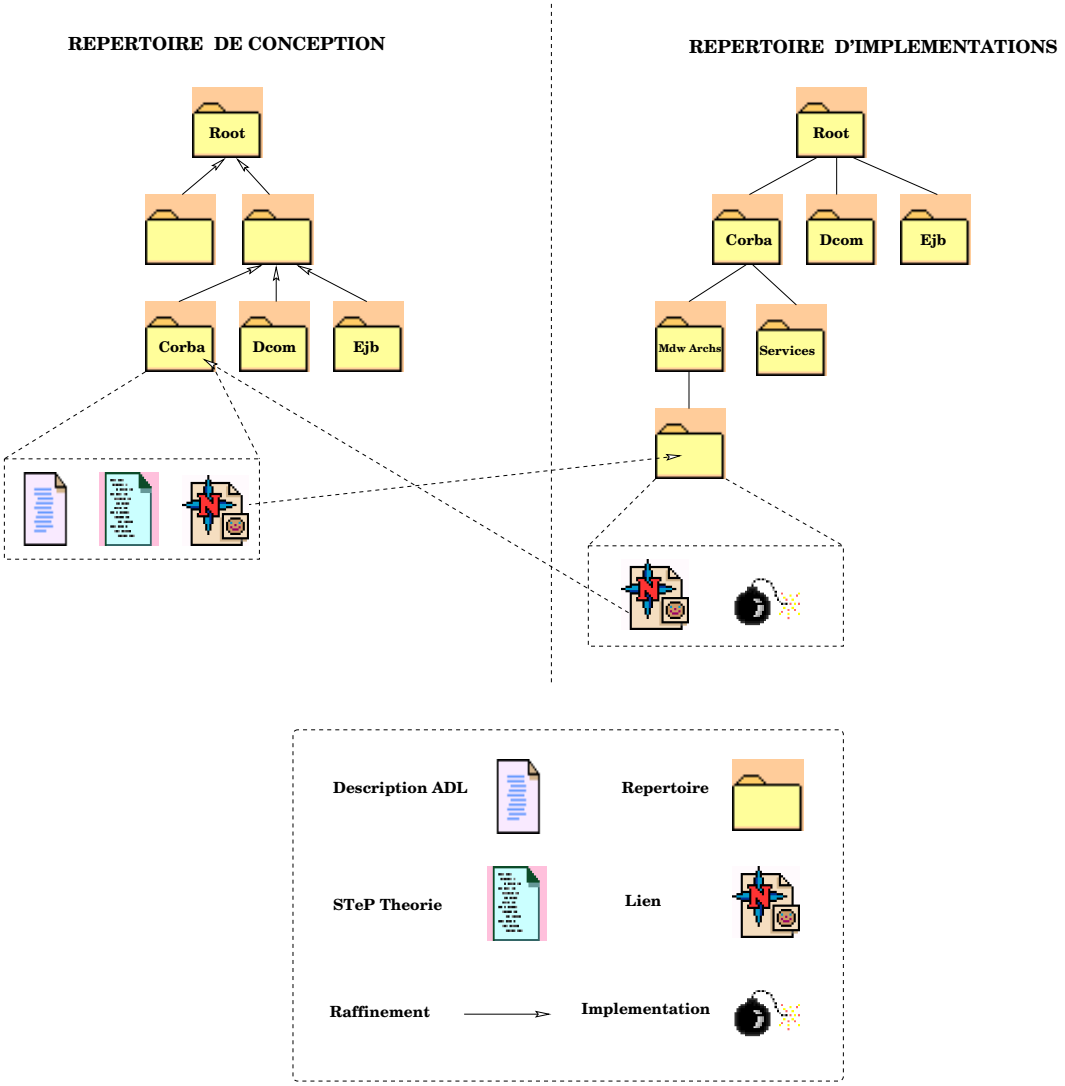

Figure III.2 Système de stockage de *middleware*

- L'un stocke les informations pour les composants qui implantent les services fournis par l'infrastructure.
- L'autre contient les implémentations des architectures *middleware*, déjà construites à partir de ces services.

Un sous-répertoire de services comprend lui-même un certain nombre de sous-répertoires, chacun étant spécifique à un service. Chaque répertoire spécifique à un service contient :

- Un fichier avec la description ADL du service,
- Un fichier qui stocke la spécification formelle des propriétés fournies par le service, et
- Un fichier donnant le code qui implante le service.

Un répertoire d'architecture stocke pour sa part différentes implémentations d'architectures *middleware* disponibles au sein de répertoires correspondants, lesquels ont également des liens vers des répertoires du système de stockage de conceptions de *middleware* qui détiennent les informations architecturales relatives à ces implémentations.

À titre d'illustration, la partie droite de la figure III.2 représente un système de stockage d'implémentations de *middleware*.

III.2.2 Simplification de la tâche de l'architecte

Étant donné le système de stockage de conceptions de *middleware*, l'architecte a à sa disposition, sous une forme structurée, toutes les tentatives antérieures de raffinement d'exigences abstraites d'applications en des architectures concrètes de *middleware*. Cette connaissance peut être une aide précieuse pour les futures tentatives de raffinements d'exigences d'applications. Toutefois, la taille du système de stockage est en général assez grande et requiert donc une méthode simple et efficace pour la recherche d'éléments au sein du système. Comme déjà indiqué, le but de l'architecte est de raffiner des exigences abstraites en une architecture concrète de *middleware*. Il s'ensuit que l'architecte doit disposer d'une méthode lui permettant de facilement localiser des séquences de raffinements conduisant à des architectures concrètes de *middleware* qui satisfont les exigences abstraites d'une application donnée. Une solution simple consiste à débuter la recherche à partir de la racine. Néanmoins, au regard de la définition de la relation de raffinement proposée dans la section III.1, cette solution nécessite de démontrer que les propriétés de l'architecture *middleware* considérée, impliquent les exigences de l'application à chaque étape du parcours du système de stockage. Réaliser cette dernière tâche de manière systématique requiert l'utilisation d'un démonstrateur de théorèmes, ce qui s'avère peu efficace et en général peu aisé.

Nous proposons par conséquent une méthode inspirée des travaux présentés dans la référence [77], ce qui permet de réduire l'utilisation d'un démonstrateur de théorèmes lors de la localisation de séquences de raffinements conduisant à des architectures concrètes de *middleware* raffinant les exigences d'une application donnée. Précisément, les propriétés fournies par une architecture *middleware* sont décrites à partir du style architectural de base, introduit dans la section II.2, ou à partir d'extensions de ce style. De la même manière, les exigences des applications sont décrites à partir de ce style ou des extensions de celui-ci. Il s'ensuit qu'une architecture *middleware* du système de stockage ne peut raffiner les exigences d'une application que si la description des propriétés qu'elle fournit est basée sur le style architectural utilisé pour décrire les exigences de l'application, ou une extension de ce style. À partir de ces remarques, la localisation d'architectures abstraites de *middleware* qui sont progressivement raffinées en des architectures concrètes qui satisfont les exigences d'une application donnée, se décompose en deux étapes :

(1) Localiser des architectures abstraites de *middleware* dont les propriétés sont décrites en utilisant au moins le même style que celui utilisé pour décrire les exigences de l'application.

(2) À partir de ces architectures abstraites, tenter de localiser une architecture qui raffine les exigences de l'application, au regard de la définition du raffinement correct d'architectures, donnée dans la section III.1.

Indiquons ici que si l'une ou l'autre des étapes précédentes échoue, alors il n'existe pas d'architecture *middleware* au sein du système de stockage de conceptions de *middleware* qui puisse être raffinée en une architecture concrète qui satisfait les exigences de l'application considérée. Il s'ensuit que l'architecte est obliger de concevoir une nouvelle architecture concrète. Pendant le processus de conception, l'architecte réalise en général plusieurs étapes de raffinement, ce qui conduit à introduire les architectures *middleware* correspondantes dans le système de stockage tout en préservant la relation de raffinement entre les éléments du système.

III.2.3 Simplification de la tâche du concepteur

Considérant un processus de développement de logiciel classique, le concepteur d'applications obtient au moins de l'architecte, un répertoire qui contient les informations relatives à une architecture concrète de *middleware*. La principale responsabilité du concepteur est alors de réaliser cette architecture, en prenant éventuellement en compte des contraintes quantitatives de mise en œuvre comme, par exemple, les performances à l'exécution ou encore l'extensibilité. La première tâche du concepteur est de vérifier si le répertoire que lui a fourni l'architecte contient des liens vers des implémentations de l'architecture concrète de *middleware* du système de stockage d'implémentations de *middleware*. Si tel est le cas, le concepteur s'assure que l'implémentation disponible satisfait les contraintes de mise en

œuvre posées. Dans l'affirmative, le travail du concepteur est terminé et le développeur d'applications prend en charge l'intégration de l'implémentation disponible du *middleware* avec l'application. Indiquons ici que notre solution à la configuration systématique de *middleware* ne fournit aucune aide au concepteur d'applications quant à la vérification des contraintes posées en termes de propriétés quantitatives comme les performances à l'exécution. La proposition de solutions à ce problème, fait partie des objectifs de l'action de recherche ASTER dans laquelle s'inscrit notre travail, et est en partie examinée dans la référence [13], qui propose pour cela des extensions du système de stockage d'implémentations de *middleware*. Le travail présenté dans cette référence introduit en outre un complément à l'approche proposée dans le chapitre suivant pour gérer l'évolution dynamique de *middleware*.

Considérant à nouveau la responsabilité du concepteur d'applications, s'il n'existe pas d'implémentation disponibles de *middleware* satisfaisant l'ensemble des exigences de l'application, le concepteur doit proposer une nouvelle implémentation de *middleware*. Ce travail requiert notamment la conception des différents éléments de l'architecture concrète correspondante, comme révélée par l'architecte. Il est toutefois possible que le système de stockage d'implémentations de *middleware* contienne des implémentations de tous, ou partie de, ces éléments. Par conséquent, avant de débuter la conception de tout élément de l'architecture, le concepteur peut rechercher cet élément dans le système de stockage. Comme le concepteur dispose de l'architecture concrète du *middleware*, il connaît les propriétés caractérisant le comportement observable de chacun des éléments constituant l'architecture. De plus, comme l'architecture *middleware* est concrète, le concepteur connaît l'infrastructure *middleware* choisi et donc s'il recherche des éléments qui implantent des services CORBA, DCOM, ou encore EJB, pour ne citer que les infrastructures les plus utilisées.

De manière générale, la recherche d'implémentations disponibles au sein du système de stockage, pour construire l'implémentation d'une architecture concrète de *middleware*, se décompose en les étapes suivantes :

(1) Sélection du répertoire de services de l'infrastructure *middleware* sur laquelle repose l'architecture concrète du *middleware*.

(2) Pour chaque composant *middleware*, à l'exception de ceux décrivant les *proxies* de l'application, nécessaire à la construction de l'architecture concrète du *middleware*, faire :

   (A) Localiser les composants *middleware* dont les propriétés sont décrites en utilisant, au mois, le même style que celui utilisé pour décrire les propriétés du composant *middleware* requis.

   (B) À partir des composants *middleware* identifiés lors de l'étape précédente, choisir ceux dont les propriétés impliquent celles du composant requis (*i.e.*, choisir les composants *middleware* qui raffinent celui requis pour construire l'architecture concrète du *middleware*).

Étant données les implémentations trouvées pour les éléments de l'architecture concrète du *middleware*, le développeur prend en charge l'implémentation effective du *middleware*. Notons ici que lorsqu'une nouvelle implémentation d'architecture concrète de *middleware* est développée, le concepteur l'insère dans le système de stockage d'implémentations suivant les règles de construction du système, énoncées précédemment.

III.3 Intégration de middleware

L'intégration d'une implémentation de *middleware* qui réalise une architecture concrète comprend :

- L'assemblage des implémentations des éléments composant l'architecture concrète du *middleware*, et
- La combinaison de l'implémentation résultante avec l'application.

Cette tâche est typiquement réalisée par les développeurs d'applications à partir de ce que leur fournissent les concepteurs. Dans le meilleur des cas, où l'architecture concrète du *middleware* a été trouvée dans le système de stockage, le développeur doit seulement composer l'implémentation du *middleware* avec l'application. Dans le pire des cas, le développeur doit construire l'architecture concrète du *middleware* à partir de services *middleware* existants, qui implantent les composants de l'architecture. Cette section propose une méthode systématique pour l'assemblage de l'implémentation d'une architecture *middleware*, qui peut être facilement composée avec toute application donnée.

III.3.1 Simplification de la tâche du développeur

Examinons plus avant le travail du développeur. Considérons le pire cas mentionné ci-avant, c'est-à-dire, supposons que le système de stockage de *middleware* ne contienne pas d'implémentation pour l'architecture concrète du *middleware*. Le concepteur de logiciel a alors fourni au développeur, un ensemble de composants implantant des services *middleware* offerts par une certaine infrastructure. À partir de notre présentation de la section III.1, le développeur doit réaliser les tâches suivantes, qui sont détaillées dans la suite de cette section :

- Construire les *proxies* clients et serveurs réalisant les connections explicites entre les services *middleware*.
- Construire les *proxies* clients et serveurs qui combinent les fonctionnalités du courtier et éventuellement celles de services, de telle sorte que les composants de l'application puissent interagir de manière à satisfaire leurs exigences vis-à-vis du système d'exécution sous-jacent.

- Construire les *wrappers* qui assemblent les *proxies* clients et combinent les fonctionnalités fournies par les services *middleware* de manière à permettre aux composants de l'application d'interagir suivant leurs exigences.

Si deux composants *middleware*, implémentant des services fournis par une infrastructure, sont explicitement connectés, le développeur doit construire les *proxies* client et serveur correspondant de manière à permettre à ces services d'effectivement interagir. La plupart des infrastructures *middleware* existantes offrent des outils qui, étant donnée une information adéquate, fournie par le développeur, génère automatiquement les implémentations de *proxies* client et serveur. Ainsi, la tâche du développeur se ramène à générer les données adéquates pour les outils de l'infrastructure. De plus, la construction de connections entre des services n'est réalisée qu'une fois, lors du premier assemblage de l'implémentation de l'architecture concrète du *middleware*. Le travail du développeur est donc trivial à chaque fois que l'implémentation est réutilisée.

Si l'on reprend notre présentation de la section III.1, un *wrapper* se divise en deux parties : le *wrapper* client et le *wrapper* serveur. Les interfaces respectivement fournies et requises par ces éléments d'un *wrapper* dépendent de l'application. Par conséquent, lorsque l'on construit l'implémentation d'une architecture concrète de *middleware* pour la première fois, le développeur doit construire la partie qui est spécifique à l'application. De plus, cette partie, dépendante de l'application, doit être reconstruite à chaque réutilisation de l'implémentation du *middleware*. Il s'avère que les parties du *middleware* qui sont dépendantes de l'application ne sont pas liées à l'implémentation de cette dernière ; elles dépendent uniquement de l'architecture de l'application (*e.g.*, types de composants, interfaces, *etc.*). Étant données ces dernières remarques et afin de simplifier le travail des développeurs, nous proposons une méthode pour la construction systématique des parties du *middleware* qui sont dépendantes de l'application. Le principe de notre solution est de fournir un langage simple au développeur, lequel permet de décrire abstraitement (*i.e.*, indépendamment d'une application particulière) le processus de construction des parties du *middleware* qui sont dépendantes de l'application. La description de ce processus est à fournir lors de la première construction de l'implémentation de tout *middleware*. Cette description du processus est alors utilisée par un générateur, en conjonction avec la description ADL de l'application, pour générer les parties du *middleware*, dépendantes de l'application à chaque fois que ce *middleware* particulier est réutilisé pour une application différente. Cette solution a initialement été introduite dans la référence [93] puis enrichie dans la référence [92].

Pour composer une implémentation de *middleware* avec une application, le développeur doit soit implanter les *proxies* clients et serveurs réalisant les interactions entre les composants de l'application, soit fournir les données correspondantes aux outils de l'infrastructure qui génèrent les implémentations de *proxies*. Les interfaces et implémentations de *proxies* sont dépendantes de l'application. Il s'ensuit que les données fournies aux outils pour leur génération automatiques sont dépendantes de l'application. Toutefois, comme précédemment, les parties du *proxy* qui sont dépendantes de l'application ne s'appuient que sur

```
Input             ::=  <TextStream>
TextStream        ::=  <TextStream> <PlainText>
                    |  <TextStream> <PlainPrefix>
                    |  '(' <TextStream> ')'
                    |  <TextStream> <Process>
PlainText         ::=  '$'
                    |  '.'
                    |  'Iff'
                    |  'Iterate'
                    |  'Not'
                    |  [ \t\n]*
                    |  Id
Process           ::=  '$' '(' <ProcessBody> ')'
ProcessBody       ::=  <ProcessObject>
                    |  <ProcessNegation>
                    |  <ProcessIteration>
                    |  <ProcessCondition>
ProcessObject     ::=  Id
                    |  <ProcessObject> '.' Id
ProcessNegation   ::=  'Not' <Process>
ProcessIteration  ::=  'Iterate' <Process> Id <TextStream>
ProcessNegation   ::=  'Iff' <Process> <TextStream>
```

Figure III.3 Grammaire du langage du processus de génération d'implémentation

l'architecture de celle-ci. Aussi, à partir de notre solution précédente, il est possible de simplifier le processus de génération des implémentations de *proxies*. En l'absence d'un outil générant automatiquement les implémentations de *proxies*, le développeur utilise le langage que nous avons proposé pour décrire abstraitement le processus d'implémentations de *proxies* pour les composants de l'application. Si, en revanche, un tel outil est disponible, le langage proposé peut être utilisé pour décrire le processus de production des données nécessaires à l'outil. Ainsi, à chaque fois que l'architecture *middleware* est réutilisée pour gérer les interactions entre les composants d'une nouvelle application, la description du processus, combinée avec la description ADL de l'application considérée, sont exploitées pour fournir les données nécessaires à un générateur de code qui produit les *proxies*.

En général, le langage proposé pour la génération des parties d'un *middleware* dépendantes de l'application, doit permettre au développeur de décrire un *processus* itératif sur les éléments architecturaux constituant l'application (*e.g.*, composants, configurations, *etc.*), ou sur les entités de base (*e.g.*, interfaces, operations, *etc.*) qui caractérisent récursivement ces éléments architecturaux. Ensuite, pour chacun des éléments ou entités significatifs,

le langage doit permettre de décrire le code source devant être généré. Ainsi, l'*objet* d'un processus qui décrit la manière de développer les parties du *middleware* qui sont dépendantes de l'application, est soit un élément architectural, soit une entité qui caractérise récursivement un tel élément architectural. Le langage de description du processus de génération d'implémentation doit également fournir le moyen au développeur de décrire un processus qui teste si certains objets du processus sont présents ou non dans la description architecturale de l'application, et qui, suivant le cas, produit une certaine fraction de code source. Par exemple, le développeur peut vouloir décrire un processus qui teste si une opération retourne ou non un résultat, et qui, suivant le cas, génère ou non la déclaration d'une variable pour le stocker. Le développeur peut aussi vouloir décrire un processus qui vérifie si un composant est composite ou non, et qui suivant le cas génère les *proxies* client et serveur correspondants. Tout objet de la description ADL d'une application est par ailleurs nommé. Aussi, le langage de processus doit fournir le moyen de décrire un processus qui génère un code source contenant des noms caractérisant un objet du processus. Par exemple, le nom d'un composant donné dans la description ADL de l'application est nécessaire pour générer un module CORBA correspondant. De la même manière, le nom d'une interface donné dans une description ADL d'application est requis pour générer l'interface correspondante dans l'IDL de CORBA. Dans le cadre de notre travail de thèse, nous avons défini un langage de processus minimal, extensible, répondant aux exigences énoncées ci-avant. La grammaire de ce langage est donnée en figure III.3. La grammaire accepte tout texte correspondant à une fraction de code source, et qui contient éventuellement des directives du processus de développement. Une directive débute toujours avec le caractère '$' et est donnée entre parenthèses. Une directive du processus peut être de quatre types différents : un objet, une itération, une condition, ou une négation.

III.4 Outils

En conclusion de ce chapitre, la figure III.4 donne une vue complète des outils servant à la synthèse systématique de *middleware*, à partir d'une description ADL des *middleware* et applications.

Nous identifions en premier lieu l'algorithme de recherche de *middleware* qui prend en entrée un fichier STeP qui décrit les exigences d'une application donnée. Cet algorithme utilise en outre des fichiers STeP stockés dans le système de stockage de conceptions de *middleware*, afin de réaliser la première étape de la recherche fondée sur des théories communes. Ces différents fichiers STeP sont ensuite utilisés par le démonstrateur de théorèmes STeP lors de la deuxième étape de la recherche, fondée sur la relation de raffinement entre architectures *middleware*.

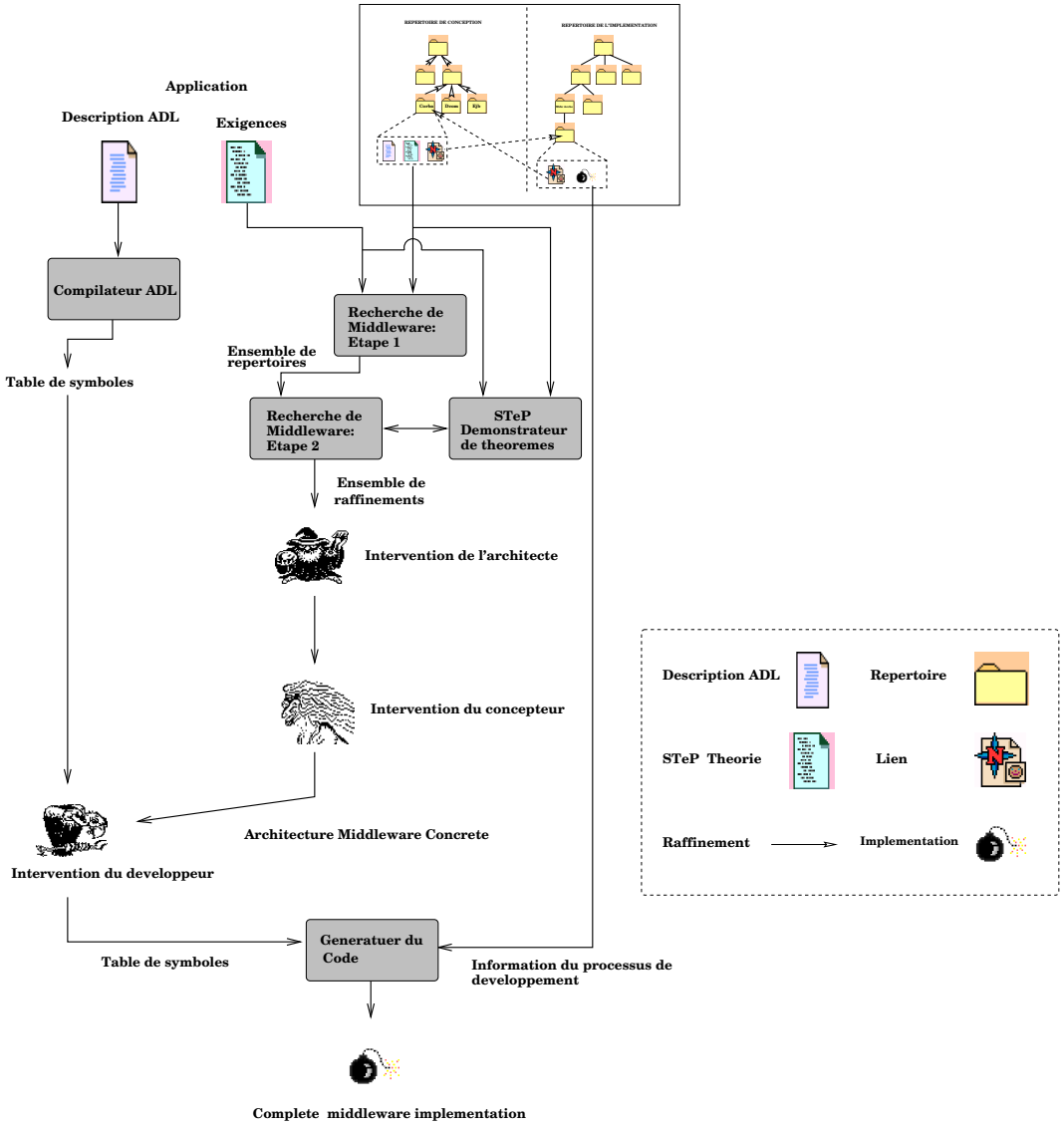

Figure III.4 Outils pour la synthèse de *middleware*

Les outils proposés comprennent également un générateur de code qui prend en entrée des informations architecturales, spécifiques à l'application et fournies par le compilateur ADL, et une architecture concrète de *middleware* incluant des directives du processus de développement. Le générateur de code assure la production des parties de l'implémentation du *middleware* qui sont dépendantes de l'application.

IV Échange de middleware

Le processus d'échange systématique de *middleware*, permettant la maintenance de ce dernier au regard de l'évolution des exigences de l'application ou de l'infrastructure (cf. § I.3), comprend :

(1) Synthétiser une nouvelle architecture *middleware* qui reflète les modifications apportées au niveau des exigences de l'application ou de l'infrastructure *middleware*.

(2) Échanger l'ancienne architecture *middleware* avec la nouvelle tout en préservant la cohérence de l'application.

La solution à la synthèse systématique de *middleware*, présentée dans le chapitre précédent, peut être utilisée pour synthétiser une nouvelle architecture *middleware* suite à l'évolution des exigences de l'application ou de l'infrastructure *middleware* (*e.g.*, modification de la disponibilité des services). Il ne nous reste donc plus qu'à proposer une stratégie pour l'échange effectif d'un *middleware* par un autre. Ce chapitre introduit une telle stratégie pour l'échange dynamique de *middleware* et précise en outre les contraintes en résultant sur la conception d'architectures *middleware* afin que celles-ci soient effectivement échangeables. Les exigences relatives à la correction de l'échange de *middleware* sont :

- Préserver la *cohérence* de l'application, c'est-à-dire, garantir que l'application reste dans un état cohérent après tout échange de *middleware*, et introduire une *perturbation minimale* dans l'exécution de l'application.

- Garantir que l'échange est *atteignable*, signifiant qu'il est réalisé en un temps fini.

Les exigences indiquées sont précisément définies par la suite, à partir du modèle de spécification introduit dans la section II.2. Ces définitions sont ensuite exploitées pour la conception d'une architecture concrète de *middleware* satisfaisant au moins les exigences identifiées ci-avant.

IV.1 Analyse et spécification des besoins

Cette section fournit une spécification formelle des propriétés relatives à l'échange correct de *middleware* en s'appuyant sur le modèle de spécification de la section II.2. Rappelons que ces propriétés relèvent du maintien de la cohérence de l'application et de l'atteignabilité de l'échange.

IV.1.1 Maintien de la cohérence

Afin de maintenir la cohérence de l'application durant l'échange du *middleware* sous-jacent, l'échange doit être réalisé :

- Lorsqu'il n'y a aucune requête en cours [44], ou
- S'il existe des requêtes en cours, lorsque l'état de l'ancien *middleware* peut être traduit en un état du nouveau *middleware* de telle sorte que les requêtes en cours verront leurs exigences vis-à-vis du *middleware* satisfaites [86].

Afin de spécifier formellement les propriétés indiquées ci-dessus, le style architectural `Base-Architectural-Style` doit être étendu de manière à permettre :

- La caractérisation des requêtes en cours (ou *pending requests*),
- La définition de fonctions de transfert d'états entre *middleware* (ou *state mapping functions*), et
- La description d'une situation d'échange particulière.

Ces extensions sont données par la théorie de nom `Exchangeable-Middleware-Style`, laquelle est utilisée pour décrire les contraintes liées à la correction de l'échange de *middleware* ainsi que les propriétés caractérisant les architectures *middleware* échangeables.

Précisément, le type $\mathcal{PENDING}$ suivant est utilisé dans les spécifications, pour définir des fonctions booléennes prenant une requête et l'identificateur unique correspondant (req, rid) en arguments, et étant évaluée à *vrai* si la requête est en cours, (*i.e.*, une propriété exigée du *middleware* n'est pas vérifiée pour req, rid), et évaluée à *faux* sinon.

$$\texttt{type}\ \mathcal{PENDING}() : \mathcal{R} * \mathcal{UID} \rightarrow \{true, false\}$$

Nous introduisons par ailleurs le type $\mathcal{MAP}$ suivant, afin de fournir le moyen de définir des fonctions traduisant l'état d'un *middleware* en l'état d'un autre *middleware*.

$$\texttt{type}\ \mathcal{MAP}() : \Sigma \rightarrow \Sigma$$

Enfin, nous définissons le type $XSituation$ suivant afin de permettre la caractérisation d'une situation d'échange donnée. Ce type est une structure qui identifie : une situation d'échange au moyen d'un identificateur unique, l'ancien *middleware*, le nouveau *middleware*, un ensemble de fonctions de type $\mathcal{PENDING}$ où chacune des fonctions est utilisée pour caractériser les requêtes en cours au regard d'une propriété particulière fournie par l'ancien *middleware*, une fonction de transfert d'état qui est nulle par défaut, et deux requêtes qui sont respectivement émises au début et à la fin de la situation d'échange.

$$\begin{array}{l}\texttt{type}\\ \quad XSituation = \{\\ \quad\quad xid : \mathcal{UID},\\ \quad\quad MdwOld : \mathcal{CONF},\\ \quad\quad MdwNew : \mathcal{CONF},\\ \quad\quad PendingFunctions : \{Pending_{Property} : \mathcal{PENDING}\} \neq \emptyset,\\ \quad\quad Map : \mathcal{MAP} = \bot,\\ \quad\quad reqXBEG : \mathcal{R},\\ \quad\quad reqXEND : \mathcal{R}\\ \quad \}\end{array}$$

La théorie `Exchangeable-Middleware-Style` introduit en outre deux *macros*, *SafeState* et *IdleState* : la première définit précisément un *état sûr* (ou *safe state*) au regard d'une situation d'échange particulière ; la seconde définit précisément un état en veille (ou *idle state*) suivant la référence [44].

Précisément, la *macro* suivante est vérifiée pour une situation d'échange donnée si l'ancien *middleware* est dans un état σ pouvant être traduit en un état du nouveau *middleware*, qui est alors capable de satisfaire les exigences de toutes les requêtes en cours au moment de l'échange :

$$\begin{array}{l}\texttt{macro}\\ SafeState(\sigma : \Sigma, MdwOld : \mathcal{CONF}, MdwNew : \mathcal{CONF},\\ \qquad Pending_{Property} : \mathcal{PENDING}, Map : \mathcal{MAP})\ \texttt{=}\\ \quad [MdwOld, \sigma] \wedge\\ \quad (\forall req : \mathcal{R},\\ \quad\quad rid : \mathcal{UID} \mid\\ \quad\quad\quad (Pending_{Property}(req, rid) \wedge \Diamond[MdwNew, Map(\sigma)]) \Rightarrow (\Diamond\Box\neg Pending_{Property}(req, rid)))\end{array}$$

De la même manière, la *macro* $IdleState$ est vérifiée si l'ancien *middleware* est dans un état σ où aucune requête n'est en cours :

$$
\begin{array}{l}
\texttt{macro} \\
IdleState(\sigma : \Sigma, MdwOld : \mathcal{CONF}, Pending_{Property} : \mathcal{PENDING}) = \\
\quad [MdwOld, \sigma] \wedge \ (\nexists req : \mathcal{R},\ rid : \mathcal{UID} \mid Pending_{Property}(req, rid))
\end{array}
$$

À partir des définitions introduites, nous obtenons que l'échange de *middleware* préserve la cohérence de l'application si la propriété `Consistency` suivante est vérifiée lors de tout échange :

$$
\begin{array}{l}
\texttt{PROPERTY Consistency} \\
\forall X : XSituation \mid \\
\quad (X.Map \neq \perp \Rightarrow \\
\quad\ (\exists C, C' : \mathcal{C} \mid \\
\quad\quad Call(X.MdwOld, C, C', X.reqXEND, X.xid)) \Rightarrow \\
\quad\ (\exists \sigma : \Sigma \mid \\
\quad\ \Diamond(\forall Pending_{Property} : \mathcal{PENDING} \mid \\
\quad\ Pending_{Property} \in X.PendingFunctions \wedge \\
\quad\ SafeState(\sigma, X.MdwOld, X.MdwNew, Pending_{Property}, X.Map)))) \wedge \\
\quad (X.Map = \perp \Rightarrow \\
\quad\ (\exists C, C' : \mathcal{C} \mid \\
\quad\quad Call(X.MdwOld, C, C', X.reqXEND, X.xid)) \Rightarrow \\
\quad\ (\exists \sigma : \Sigma \mid \\
\quad\ \Diamond(\forall Pending_{Property} : \mathcal{PENDING} \mid \\
\quad\ Pending_{Property} \in X.PendingFunctions \wedge \\
\quad\ IdleState(\sigma, X.MdwOld, Pending_{Property}))))
\end{array}
$$

La propriété `Consistency` spécifie que s'il existe une fonction de transfert d'état pour un échange donné, l'échange peut survenir alors qu'il existe des requêtes en cours, sinon l'échange survient lorsqu'il n'y a aucune requête en cours comme cela est suggéré par la solution classique à la reconfiguration dynamique [44].

IV.1.2 Atteignabilité d'un état sûr

Atteindre un état sûr du *middleware* en un temps fini ne peut être garanti sans imposer des contraintes sur le comportement de l'application [44]. Une exigence plus pragmatique vis-à-vis du processus d'échange de *middleware* est donc de guider l'atteinte d'un état sûr. Ceci peut être réalisé en bloquant sélectivement les composants de l'application de manière à les empêcher d'émettre des requêtes éloignant le *middleware* d'un état sûr. Par exemple, considérant les composants d'une application interagissant *via* un *middleware* transactionnel, les seuls composants ne devant pas être bloqués sont ceux dont l'exécution est nécessaire à la terminaison des transactions en cours. De la même manière, considérant

des composants d'une application qui interagissent *via* un *middleware* sécurisant les communications, les seuls composants ne devant pas être bloqués sont ceux dont l'exécution est nécessaire à la terminaison d'une session sécurisée.

Analysons plus avant les exigences que nous avons mentionnées ci-avant, quant à la correction de l'échange de *middleware*. Il est important de remarquer que la granularité du blocage des composants a une incidence significative sur l'efficacité du processus de blocage. Par définition, un composant est une unité de calcul ou de stockage de données. Cette définition est plutôt générale et ne donne aucun détail quant à la manière dont un calcul est effectué par un composant donné. Par exemple, la définition donnée pour un composant n'indique pas si un composant exécute des activité concurrentes (également qualifié de *mutli-threaded* en anglais). Dans l'affirmative, un composant dont l'exécution est considérée comme nécessaire pour atteindre un état sûr, peut voir toutes ses activités maintenues actives si le blocage est réalisé au niveau du composant. Néanmoins, seulement un sous-ensemble des activités d'un composant doivent typiquement être maintenues actives pour permettre au *middleware* d'atteindre un état sûr. Considérons par exemple un composant à activités concurrentes qui initie une transaction. Supposons en outre que ce soit ce même composant qui termine la transaction *via* l'exécution de l'une de ses activités. Dans ce cas précis, seule l'activité terminant la transaction doit poursuivre son exécution. Les autres activités du composant peuvent et doivent être bloquées puisqu'elles empêchent le *middleware* d'effectivement atteindre un état sûr, soit en émettant des requêtes *via* le *middleware* ou en consommant de la ressource processeur aux dépens de l'activité pouvant conduire le *middleware* dans un état sûr. En résumé de ce paragraphe, il s'avère que la manière dont sont exécutés les calculs a une incidence significative sur la granularité du blocage des exécutions de composants, et par conséquent sur l'efficacité du processus de blocage.

Au regard des remarques que nous avons faites jusqu'ici, le processus de blocage doit sélectivement bloquer les activités des composants de l'application de manière à les empêcher d'émettre des requêtes prévenant l'atteinte d'un état sûr par le *middleware*. Avant de donner les spécification formelle de cette propriété, deux points restent à traiter :

- Étendre la définition de `Base-Architectural-Style` de manière à intégrer la définition des activités exécutées par les composants.
- Étendre la définition de `Base-Architectural-Style` de manière à associer les activités avec les requêtes qu'elles émettent.

Comme déjà indiqué, la théorie résultant de cette extension est appelée `Multithreaded--Architectural-Style`.

Définition des activités d'exécution. À partir d'ici et ce, pour le reste de ce document, nous supposons que tout calcul est effectué pour le compte d'un composant, par

un ensemble d'activités s'exécutant concurremment. Cet ensemble contient au moins une activité. Les activités d'un composant n'ont pas d'état associé mais partagent et manipulent l'état du composant. Les activités peuvent être créées, détruites, et leur exécution synchroniser en utilisant les fonctionnalités offertes par le *middleware* sous-jacent. Une activité peut être active (*i.e.*, du fait de l'exécution d'un calcul) ou en veille. Un activité est caractérisée par un ensemble de variables (*e.g.*, compteur ordinal) dont la valeur est sauvegardée à chaque fois que l'activité est desactivée et restaurée à chaque fois que l'activité est de nouveau activée. Ces variables font partie de l'état du composant auquel appartient l'activité considérée.

Dans la section II.2, nous avons défini Σ comme un type utilisé dans les spécifications pour définir les états des composants. À cette occasion, Σ a été défini comme un ensemble de variables dont un sous-ensemble comprend à présent des informations relatives aux activités du composant. Posons `type` $\mathcal{THR}$, défini par la théorie `Multithreaded-Architectural-Style`, comme étant le type utilisé pour la définition des activités (ou *threads*). Nous obtenons alors que l'axiome suivant est vérifiée :

`AXIOM Threads`
$$\mathcal{THR} \subset \Sigma$$

Nous avons en outre la définition de la fonction *ActiveThread* suivante par la théorie `Multithreaded-Architectural-Style`. Cette fonction prend une instance de composant en argument et retourne la partie de l'état du composant relative à l'activité qui est en cours d'exécution. Indiquons que l'introduction de cette fonction n'est correcte que si nous considérons l'exécution de tout composant au-dessus d'une machine mono-processeur. Dans le cas d'une machine multi-processeur, plusieurs activités peuvent effectivement s'exécuter concurremment, conduisant à une fonction *ActiveThread* retourne un ensemble d'activités et non pas une seule activité.

$$\texttt{value}\ \mathit{ActiveThread} : \Sigma \rightarrow \mathcal{THR}$$

Comme précédemment indiqué, les activités peuvent être créées, détruites et leur exécution synchroniser en utilisant les fonctionnalités du *middleware* sous-jacent. En particulier, lors de la synchronisation de deux activités t, t', nous disons que ces activités sont *dépendantes* et ce, depuis la requête de synchronisation jusqu'à la synchronisation effective des deux activités. Par exemple, si t attend un signal de t', il se peut qu'il existe des requêtes qui ne seront émises par t qu'après que t ait reçu le signal de t'. Par conséquent, si t' n'atteint pas le point de synchronisation où elle émet le signal à t, t ne pourra pas poursuivre son exécution. Considérons à présent le cas où deux activités t, t', du même composant C, synchronisent leur exécution *via* un mécanisme de barrière. Si l'une de ces deux activités n'atteint pas le point de barrière, aucune d'elles ne pourra s'exécuter au delà de la barrière. Il s'ensuit que l'exécution de t dépend de l'exécution de t' jusqu'à ce que la barrière soit atteinte. Ceci nous amène à introduire la définition de la fonction *dep*

suivante, au sein de `Multithreaded-Architectural-Style`, de manière à permettre la caractérisation de dépendances entre activités. Cette fonction prend deux activités t, t' en argument et est évaluée à *vrai* tant que l'exécution de t dépend de celle de t', et est est évaluée à *faux* sinon.

$$\texttt{value}\ dep : \mathcal{THR} * \mathcal{THR} \rightarrow \{true, false\}$$

Notons que dans le cas où les activités synchronisent leur exécution *via* un mécanisme de barrière et tant que la barrière n'est pas atteinte, les évaluations de $dep(t, t')$ et $dep(t', t)$ sont toutes deux évaluées à *vrai*.

Association des activités d'exécution avec leurs requêtes. Afin d'associer les activités d'exécution avec les requêtes qu'elles émettent ou reçoivent, nous devons redéfinir les *macros Call*, *UpCall*, *ReturnCall*, et *ReturnUpCall* que nous avons définies dans la section II.2. Avant d'en donner leur nouvelle définition, nous devons d'abord introduire les fonctions suivantes :

- La fonction *CallThread* est utilisée dans les spécifications pour associer une requête émise par un élément architectural à un certain instant, avec l'activité l'ayant effectivement émise. Précisément, la fonction *CallThread* prend une requête $req : \mathcal{R}$ et l'identificateur unique correspondant $rid : \mathcal{UID}$ en arguments, et retourne l'activité $t : \mathcal{THR}$ ayant émis la requête considérée :

$$CallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

- De la même manière, la fonction *UpCallThread* est utilisée dans les spécifications pour associer une requête reçue par un élément architectural avec l'activité prenant en charge le traitement de cette requête. La fonction *UpCallThread* prend une requête $req : \mathcal{R}$ et l'identificateur unique correspondant $rid : \mathcal{UID}$ en arguments, et retourne l'activité associée $t : \mathcal{THR}$:

$$UpCallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

- La fonction *ReturnCallThread* est pour sa part utilisée dans les spécifications pour associer à la réponse d'une requête émise par un élément architectural, l'activité qui a effectivement reçu la réponse. La fonction *ReturnCallThread* prend une réponse $resp : \mathcal{R}$ et l'identificateur unique correspondant $rid : \mathcal{UID}$ en arguments, et retourne l'activité $t : \mathcal{THR}$ ayant reçu la réponse :

$$ReturnCallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

- Nous utilisons enfin la fonction *ReturnUpCallThread* dans les spécifications pour associer à une réponse émise par un élément architectural, l'activité ayant émis cette réponse. La fonction *ReturnUpCallThread* prend une réponse $resp : \mathcal{R}$ et l'identificateur unique correspondant $rid : \mathcal{UID}$ en arguments, et retourne l'activité $t : \mathcal{THR}$ correspondante :

$$ReturnUpCallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

À partir des définitions introduites ci-avant, nous sommes en mesure de redéfinir les *macros* relatives aux interactions entre les éléments architecturaux, de manière à prendre en compte les propriétés inhérentes à l'échange de *middleware*.

- La définition de *Call* est similaire à celle introduite dans le chapitre précédent à l'exception de l'introduction d'une nouvelle clause. Cette clause associe à une requête req, émise par un composant C vers un composant C', l'activité ayant effectivement émis la requête. Ceci est effectué en posant que la valeur retournée par la fonction $CallThread(req, rid)$ est égale à la valeur retournée par la fonction $ActiveThread(\sigma)$:

$$\begin{array}{l}
\texttt{macro} \\
\quad Call(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad \exists o : \mathcal{O}_r, \\
\qquad ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad ((ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in ir) \wedge \\
\qquad\quad (req \in rang(o)) \wedge \ominus[C, \sigma] \wedge req = o(\sigma, rid) \wedge \\
\qquad\quad CallThread(req, rid) = ActiveThread(\sigma))
\end{array}$$

- De la même manière, la nouvelle définition de *UpCall* diffère de sa définition précédente par l'introduction d'une nouvelle clause. Cette clause associe à une requête req, reçue par un composant C', l'activité ayant effectivement pris en charge le traitement de la requête. Ceci est réalisé en posant la valeur retournée par la fonction $UpCallThread(req, rid)$ comme étant égale à la valeur retournée par la fonction $ActiveThread(\sigma)$:

$$
\begin{array}{l}
\texttt{macro} \\
\quad UpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad (\exists ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad o : \mathcal{O}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad (ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in \sharp(I_p)C') \wedge \\
\qquad\quad ((\sigma, req, rid) \in dom(o)) \wedge \\
\qquad\quad \Diamond Call(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge [C', o(\sigma, req, rid)] \wedge \\
\qquad\quad UpCallThread(req, rid) = ActiveThread(\sigma))
\end{array}
$$

- Toujours de la même façon, la nouvelle définition de *ReturnCall* est enrichie d'une clause supplémentaire qui affecte à la valeur retournée par la fonction *ReturnCall-Thread(req, rid)*, la valeur retournée par la fonction $ActiveThread(\sigma)$:

$$
\begin{array}{l}
\texttt{macro} \\
\quad ReturnCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad \exists \sigma : \Sigma \mid \\
\qquad\quad \Diamond ReturnUpCall(Conf, C, C', resp, rid) \wedge \ominus[C, \sigma] \wedge [C, response(\sigma, resp, rid)] \wedge \\
\qquad\quad ReturnCallThread(req, rid) = ActiveThread(\sigma)
\end{array}
$$

- Enfin, la définition de *ReturnUpCall* est également enrichie d'une clause supplémentaire qui affecte à la valeur du résultat de $ReturnUpCallThread(req, rid)$, celle du résultat de la fonction $ActiveThread(\sigma)$:

$$
\begin{array}{l}
\texttt{macro} \\
\quad ReturnUpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad \exists req : \mathcal{R}, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad \Diamond UpCall(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge resp = response(\sigma, rid) \wedge \\
\qquad\quad ReturnUpCallThread(req, rid) = ActiveThread(\sigma)
\end{array}
$$

À partir de la révision des définitions du chapitre précédent, donnée par `Multithreaded-Architectural-Style`, nous sommes à même de définir formellement la propriété devant être préservée pour aider à atteindre un état sûr, du point de vue de l'échange de *middleware*. La définition de cette propriété s'appuie sur un type appelé $\mathcal{SEQ}$, défini par la théorie `Necessary-Threads` donnée ci-après . Le `type` $\mathcal{SEQ}$ est un tableau de requêtes

dont le dernier élément a toujours pour valeur nulle ($\perp$), et est utilisé dans les spécifications pour déclarer des séquences de requêtes. De plus, la fonction $filter$, elle-même définie par `Necessary-Threads`, est une fonction qui prend une séquence de requêtes *seq* et une activité t en arguments, et qui retourne une séquence de requêtes égale à *seq* de laquelle on a retranché les requêtes émises par l'activité t et par les activités dépendant de t. Enfin, la *macro* $UnNecessary$ est vérifiée pour une activité t donnée si t peut être bloquée pendant le processus d'échange de *middleware*, c'est-à-dire, si cette activité n'a pas à être exécutée pour amener la *middleware* dans un état sûr. Nous obtenons la définition suivante pour la théorie `Necessary-Threads` :

```
THEORY Necessary-Threads

include Multithreaded-Architectural-Style
include Exchangeable-Middleware-Style

type
```

$\mathcal{SEQ}$ `= array[1..] of` $\mathcal{R} * \mathcal{UID}$

`value` $filter() : \mathcal{SEQ} * \mathcal{THR} \rightarrow \mathcal{SEQ}$

`AXIOM FiniteSequence:`

$$\begin{array}{l}\forall seq : \mathcal{SEQ} \mid \\ \quad (\exists i : int \mid \\ \quad\quad i \geq 1 \wedge \\ \quad\quad (\forall j : int \mid j \geq i \wedge seq[j] = \perp) \wedge \\ \quad\quad (\forall k : int \mid k < i \wedge seq[k] \neq \perp))\end{array}$$

`AXIOM Filtering:`

$$\begin{array}{l}\forall seq, seq' : \mathcal{SEQ}, \\ t : \mathcal{THR} \mid \\ \quad seq' = filter(seq, t) \Rightarrow \\ (\forall i : int \mid \\ \quad\quad (i \geq 1 \wedge seq[i] \neq \perp \wedge CallThread(seq[i]) \neq t \wedge \neg dep(CallThread(seq[i]), t)) \Rightarrow \\ \quad\quad\quad (\exists j : int \mid j \geq 1 \wedge seq'[j] \neq \perp \wedge seq'[j] = seq[i])) \wedge \\ (\forall i : int \mid \\ \quad\quad (i \geq 1 \wedge seq'[i] \neq \perp) \Rightarrow \\ \quad\quad\quad (\exists j : int \mid j \geq 1 \wedge seq[j] \neq \perp \wedge \neg dep(CallThread(seq[j]), t) \wedge seq'[i] = seq[j]))\end{array}$$

`macro`

$CallSEQ(Conf : \mathcal{CONF}, seq : \mathcal{SEQ})$ =
  $(\exists C, C' : \mathcal{C} \mid$
    $Call(Conf, C, C', \sharp(\mathcal{R})seq[1], \sharp(\mathcal{UID})seq[1]))\wedge$
    $(\forall i, j : int \mid$
      $i \geq 1 \wedge seq[i] \neq \perp \wedge j \geq 1 \wedge seq[j] \neq \perp \wedge j > i\wedge$
      $(\exists C_i, C'_i, C_j, C'_j : \mathcal{C} \mid$
        $\Diamond(Call(Conf, C_i, C'_i, \sharp(\mathcal{R})seq[i], \sharp(\mathcal{UID})seq[i]) \Rightarrow$
          $\Diamond Call(Conf, C_j, C'_j, \sharp(\mathcal{R})seq[j], \sharp(\mathcal{UID})seq[j])))))$

macro
$UnNecessary(Conf : \mathcal{CONF}, t : \mathcal{THR}, X : XSituation)$ =
  $\forall seq : \mathcal{SEQ} \mid$
    $CallSEQ(Conf, seq) \Rightarrow$
      $\Diamond(\exists \sigma : \Sigma \mid$
        $\forall Pending_{Property} : \mathcal{PENDING} \mid$
          $Pending_{Property} \in X.PendingFunctions\wedge$
          $SafeState(\sigma, X.MdwOld, X.MdwNew, Pending_{Property}, X.Map))) \Rightarrow$
    $\exists seq' : \mathcal{SEQ} \mid$
      $seq' = filtered(seq, t)\wedge$
        $(CallSEQ(Conf, seq') \Rightarrow$
      $\Diamond(\exists \sigma : \Sigma \mid$
        $\forall Pending_{Property} : \mathcal{PENDING} \mid$
          $Pending_{Property} \in X.PendingFunctions\wedge$
          $SafeState(\sigma, X.MdwOld, X.MdwNew, Pending_{Property}, X.Map)))$

END

Nous avons alors que la contrainte associée à l'échange de *middleware* est que seules les activités dont l'exécution est nécessaire pour garantir l'atteinte d'un état sûr par le *middleware* sont autorisées à émettre des requêtes au travers du *middleware* pendant l'échange de *middleware*. Formellement, nous obtenons la définition de la propriété `Reachability` correspondante :

PROPERTY Reachability
$\forall X : XSituation,$
$req : \mathcal{R},$
$rid : \mathcal{UID} \mid$
  $\exists C, C', C'', C''' : \mathcal{C} \mid$
    $Call(X.MdwOld, C, C', X.reqXBEG, X.xid)\wedge$
    $\Diamond(Call(Conf, C'', C''', req, rid)\wedge$
    $\Diamond Call(X.MdwOld, C, C', X.reqXEND, X.xid))) \Rightarrow$
  $\neg UnNecessary(Conf, CallTread(req, rid), X)$

IV.2 Conception de l'architecture middleware

Étant données les contraintes relatives au processus d'échange systématique de *middleware* que nous avons introduites dans la section précédente, nous abordons ici la conception générale d'une architecture concrète de *middleware* satisfaisant ces contraintes. Le style architectural en résultant a initialement été introduit dans la référence [86]. À l'exception des éléments architecturaux de base, utilisés pour la conception de toute architecture *middleware* gérant les interactions entre les composants d'une application, une architecture *middleware* échangeable doit en outre intégrer des éléments architecturaux capables de :

- Détecter si le *middleware* est dans un état cohérent ou non. Comme détaillé dans la section IV.1, supporter des modifications arbitraires au niveau des exigences de l'application ou de l'infrastructure *middleware*, requiert des éléments architecturaux détectant l'atteinte d'un état sûr par le *middleware* qui soient capables de s'adapter à la situation particulière de l'échange. Par conséquent, ces éléments doivent eux-mêmes être *échangeables*.
- Bloquer les activités d'exécution qui ne sont pas nécessaire à l'atteinte d'un état sûr par le *middleware*. Les éléments architecturaux en charge du processus de blocage ne dépendent pas de la situation particulière de l'échange et cette partie de l'architecture *middleware* est par conséquent *statique*.
- Réaliser l'échange de *middleware*, c'est-à-dire, charger le nouveau *middleware* à la place de l'ancien. L'élément architectural réalisant effectivement l'échange est appelé le *coordinateur*, et est en outre en charge d'adapter les éléments détectant l'état sûr relativement à l'échange de *middleware*, en fonction de l'échange particulier effectué.

La figure IV.1 donne une vision globale d'une architecture *middleware* échangeable. Chacun des éléments architecturaux utilisés pour la construction d'une telle architecture est détaillé plus avant dans la suite de cette section.

IV.2.1 Détection d'un état sûr

Étant donnée la définition d'un état en veille pour un *middleware*, donnée par la *macro* *IdleState*, introduite en section IV.1.1, il est évident que la détection d'un tel état n'est en rien dépendante de la situation particulière de l'échange. La détection d'un tel état se ramène à analyser les interactions entre les composants de l'application et le *middleware* de manière à identifier s'il y a ou non des requêtes en cours. Par exemple, un mécanisme qui détecte l'atteinte d'un état en veille pour un *middleware* qui fournit des communications fiables, analyse simplement les requêtes émises et reçues par les composants de l'application, et rapporte que le *middleware* est dans un état en veille si pour toute requête émise, une réponse a été délivrée à l'émetteur. De la même manière, un mécanisme détectant un état

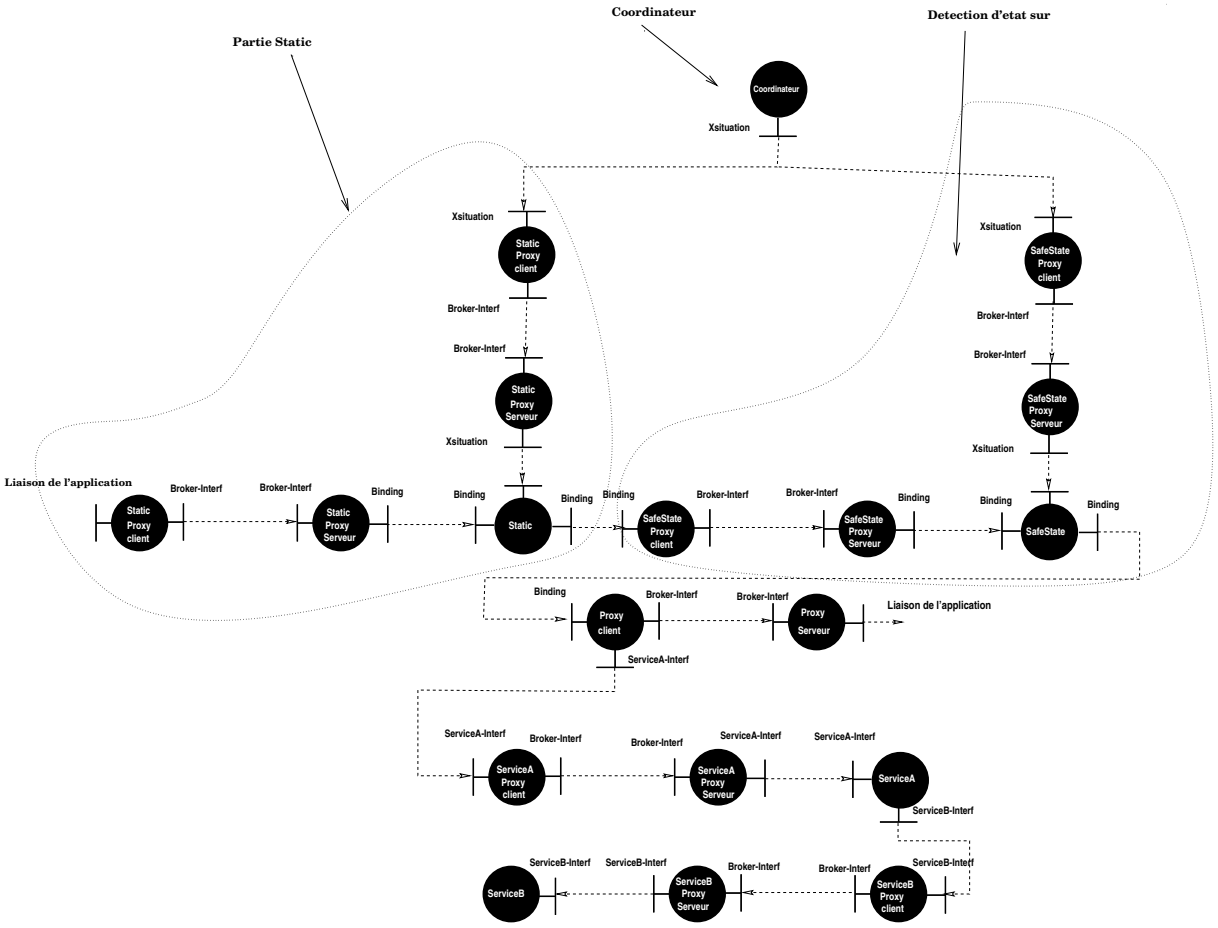

Figure IV.1 Une architecture *middleware* échangeable

en veille pour un *middleware* transactionnel analyse simplement les requêtes relatives à l'initialisation et à la terminaison de transactions. Ce mécanisme détecte alors un état en veille si pour toute requête d'initialisation de transaction, il existe une requête de terminaison de transaction correspondante. Comme la détection d'un état en veille du *middleware* est indépendante de la situation d'échange, les mécanismes correspondants peuvent aisément être réutilisés par toute architecture *middleware* offrant les mêmes propriétés. Afin de promouvoir plus avant la réutilisation de logiciel, nous considérons la définition d'un mécanisme de détection d'état en veille par propriété garantie par le *middleware*. Ceci signifie que pour un *middleware* offrant un ensemble de propriétés, autant de mécanismes que de propriétés sont utilisés pour la détection de l'état en veille. Par exemple, pour un middleware offrant des interactions transactionnelles et sécurisés, nous aurons deux mécanismes de détection d'un état en veille, respectivement relatifs à la propriété transactionnelle et à la propriété de sécurisation. Concrètement, lors de la synthèse d'un *middleware* que l'on veut échangeable, les mécanismes de détection d'un état en veille associés aux différentes propriétés fournies par le middleware, sont intégrés à l'architecture *middleware*. Un mécanisme

de détection d'un état en veille consiste en un ensemble de composants *middleware* qui analysent les interactions entre les composants de l'application et le *middleware*. Chacun de ces composants est un *wrapper* lié à un composant de l'application ou à un service *middleware* qui est explicitement utilisé par l'application. Cet ensemble de *wrappers* doit alors être coordonné de manière à détecter si le *middleware* est dans un état en veille, à partir des différents résultats d'analyse des *wrappers*. Ceci est de la responsabilité du *coordinateur* du processus d'échange de middleware, qui est détaillé plus loin dans cette section. Une solution alternative est d'introduire un coordinateur pour chaque ensemble de *wrappers* réalisant un mécanisme de détection d'un état en veille pour une propriété donnée, ou d'introduire des interactions entre les *wrappers* à des fins de coordination. Néanmoins, ces solutions complexifie la conception de l'architecture *middleware* et sont au détriment des performances à l'exécution. Signalons toutefois que bien que non abordé dans ce document, le coordinateur peut être implantée de manière distribuée si cela s'avère nécessaire, par exemple à des fins de tolérance aux défaillances.

De manière alternative à la détection d'un état en veille pour le middleware, préalablement à son échange effectif, des détecteurs d'états sûrs –si disponibles– peuvent être utilisés. De tels détecteurs peuvent minimiser de manière significative les perturbations provoquées lors de l'exécution de l'application. Toutefois, ils sont spécifiques à chaque situation d'échange et doivent être installés lorsque la situation correspondante survient. Il s'ensuit que leur disponibilité ne peut être systématiquement acquise. En général, leur utilisation permet de faire face aux cas où l'atteinte d'un état en veille du *middleware* provoque une perturbation ne pouvant être tolérée par l'application (*e.g.*, non respect d'échéances lors de l'exécution d'une application temps-réelle). La détection d'un état sûr se ramène à vérifier si l'état courant du *middleware* peut être traduit en un état correspondant du nouveau *middleware*, lequel étant alors capable de satisfaire les exigences des requêtes en cours. L'état du *middleware* évolue en fonction des interactions entre l'application et le *middleware*, et entre les éléments architecturaux du *middleware*. Il s'ensuit qu'un état sûr du *middleware* peut être détecté en analysant les interactions susmentionnées. La conception d'un mécanisme pour la détection d'un état sûr est alors similaire à celle d'un mécanisme pour la détection d'un état en veille. Le détecteur d'un état sûr comprend un ensemble de *wrappers* pour les composants clients de l'application et pour l'ensemble des composants de l'architecture *middleware*. Une solution alternative à la détection d'un état sûr consiste à directement analyser l'état du *middleware*. Remarquons que la capacité à analyser l'état du *middleware* est en général facilitée par les infrastructures *middleware* réflexives. Par exemple, dans la référence [14], une telle infrastructure est proposée, laquelle facilite l'analyse de l'état du *middleware* en exploitant le modèle, qualifié de *méta*, des ressources utilisées. Dans ce cadre, un mécanisme détectant un état sûr du *middleware* comprend des composants qui accède à l'état *réifié* du *middleware*. Installer un nouveau détecteur d'état sûr avant l'échange de *middleware* constitue également un point intéressant et dépend de si le détecteur en question analyse les interactions entre composants ou directement l'état du *middleware*. Afin de correctement détecter un état sûr, le détecteur doit accéder des informations relatives aux requêtes en cours. Ceci est possible si le détecteur est capable

d'analyser l'état du *middleware*. En revanche, dans la négative où le détecteur analyse les interactions entre composants, l'adaptation du *middleware* afin d'intégrer le détecteur, ne peut prendre place que lorsque le détecteur couramment installé rapporte qu'il n'y a aucune requête en cours. Remarquons que cette attente est moins source d'inefficacité que l'attente de l'atteinte d'un état en veille du *middleware*. Dans le dernier cas, tous les détecteurs d'un état en veille doivent simultanément rapporter un état en veille du *middleware*.

À partir d'ici, le terme de détecteur d'état sûr est employé pour signifier, aussi bien un mécanisme détectant un état sûr, qu'un mécanisme de détection d'un état en veille. En plus des interfaces requises et fournies par un détecteur d'état sûr pour analyser les interactions ou l'état du *middleware*, le détecteur fournit une interface utilisée par le coordinateur pour notifier au détecteur le début et la terminaison d'une situation d'échange particulière.

Finalement, les théories qui décrivent les propriétés d'un détecteur d'état sûr sont toutes basée sur le schéma de théorie suivant :

```
THEORY SafeState-Detection

include Exchangeable-Middleware-Style

value Pending_Property : PENDING
value Detector : C


AXIOM Report-Safe-State
∀X : XSituation |
   (∃C : C,
   σ : Σ |
     ⬙(SafeState(σ, X.MdwOld, X.MdwNew, Pending_Property, X.Map)∧
     ⬙UpCall(X.MdwOld, C, Detector, X.reqXBEG, X.xid)) ⇔
     ReturnUpCall(X.MdwOld, C, Detector, X.reqXBEG, X.xid)

AXIOM Set-Pending-True-Condition
      % Définit les conditions sous lesquelles
      % Pending_Property(req, rid)
      % est évaluée à vrai.

AXIOM Set-Pending-False-Condition
      % Définit les conditions sous lesquelles
      % Pending_Property(req, rid)
      % est évaluée à faux
END
```

La théorie donnée ci-dessus est définie à partir de `Exchangeable-Middleware-Style`. Cette théorie définit l'axiome `Report-Safe-State` qui spécifie que si un détecteur d'état sûr

est notifié du début d'une situation d'échange et que si un état sûr est atteint, alors il rapporte que cet état est atteint. De plus, les axiomes `Set-Pending-True-Condition` et `Set-Pending-False-Condition` donnent les conditions sous lesquelles une requête n'est plus considérée comme en cours ; des exemples précis de telles conditions sont donnés dans la seconde partie de ce document au travers d'une étude de cas.

IV.2.2 Blocage sélectif des exécutions

La partie statique de la gestion de l'échange de *middleware* implante le processus de blocage des exécutions. Comme déjà indiqué, ce processus est indépendant de la situation particulière d'échange considérée. Par conséquent, cette partie à intégrer à l'architecture *middleware* n'a pas à être échangeable. Il s'ensuit que la partie statique du processus d'échange de *middleware* est intégrée à chaque implémentation de *middleware* durant la phase de synthèse et reste inchangée pendant toute la durée de vie de l'application. Afin de bloquer les activités émettant des requêtes, dont l'exécution n'est pas nécessaire pour atteindre un état sûr du *middleware*, les interactions entre les composants de l'application *via* le *middleware* doivent être analysées. Par conséquent, la partie statique consiste en un ensemble de composants qui sont des *wrappers* des composants clients de l'application. À l'exception des interfaces typiquement fournies (respectivement requises) par un *wrapper*, les *wrappers* composant la partie statique de la gestion de l'échange de *middleware* fournissent une interface utilisée par le coordinateur pour être notifiés de l'initialisation et de la terminaison d'un échange donné. En particulier, la notification de l'initialisation d'un échange signifie aux *wrappers* d'installer les éventuels détecteurs d'états sûrs qui sont spécifiques à la situation d'échange considérée.

Lors de la terminaison du processus d'échange, le coordinateur signifie à la partie statique de la gestion de l'échange du *middleware* de charger le nouveau *middleware*, en remplacement de l'ancien. Dans le contexte de ce document, nous supposons que cette mise à jour est réalisée en retirant l'ancienne implémentation du *middleware* puis en chargeant l'implémentation du nouveau *middleware*. Néanmoins, les infrastructures *middleware* réflexives comme celle que nous avons précédemment référencée fournissent des fonctionnalités permettant des modifications plus modulaires. Par exemple, dans la référence [15], la modification d'une partie du *middleware* est simplifiée *via* l'accès au modèle, qualifié de *méta*, de composition du *middleware*. La disponibilité d'une telle fonctionnalité doit encourager des échanges de *middleware* modulaires. Toutefois, si la nouvelle architecture middleware s'avère être très différente de la nouvelle, la modularisation de l'échange n'est pas utile.

Finalement, chacun des *wrappers* composant la partie statique de la gestion de l'échange de *middleware* est capable de collecter les dépendances entre les activités de l'application. Étant données ces dépendance et si une situation d'échange survient, chacun des *wrappers* bloque sélectivement les activités tentant d'émettre des requêtes *via* le *middleware* tout en garantissant l'atteignabilité d'un état sûr.

La théorie `Static Support` qui suit décrit les propriétés des composants constituant la partie statique de la gestion de l'échange de *middleware*. Cette théorie repose sur la théorie `Exchangeable-Middleware-Style` et définit un axiome qui stipule que lors d'un échange, seules les activités dont l'exécution est nécessaire à l'atteinte d'un état sûr du *middleware* peuvent émettre des requêtes *via* celui-ci :

```
THEORY Static Support

include Exchangeable-Middleware-Style

AXIOM Blocking-Process
```

$$
\begin{array}{l}
\forall X : XSituation, \\
req : \mathcal{R}, \\
rid : \mathcal{UID} \mid \\
\quad (\exists C, C', C'', C''' : \mathcal{C} \mid \\
\qquad Call(X.MdwOld, C, C', X.reqXBEG, X.xid) \wedge \\
\qquad \Diamond(Call(Conf, C'', C''', req, rid) \wedge \\
\qquad \Diamond Call(X.MdwOld, C, C', X.reqXEND, X.xid)) \Rightarrow \\
\quad \neg UnNecessary(Conf, CallTread(req, rid), X))
\end{array}
$$

```
END
```

IV.2.3 Coordination

Le coordinateur est le dernier élément architectural nécessaire à la gestion de l'échange de *middleware*. Il synchronise l'ensemble du processus d'échange. En particulier, le coordinateur notifie la partie statique de la gestion de l'échange que les détecteurs d'état sûr, spécifiques à l'échange considéré, doivent être chargés. De plus, le coordinateur notifie les détecteurs d'état sûr du début du processus d'échange et leur demande de notifier l'atteinte d'un état sûr dès que cela est détecté. Enfin, lorsque tous les détecteurs d'état sûr ont rapporté que le *middleware* est dans un état sûr, le coordinateur notifie la partie statique de la gestion de l'échange de réaliser effectivement l'échange de l'ancien *middleware*.

La théorie `Coordinator` qui suit décrit les propriétés du coordinateur. L'axiome `End--Exchange` spécifie que le coordinateur termine le processus d'échange de *middleware* une fois que tous les détecteurs ont notifié que le *middleware* a atteint un état sûr :

```
THEORY Coordinator

include Exchangeable-Middleware-Style

value StaticSupport : {C : C} ≠ ∅
value SafeStateDetectors : {C : C} ≠ ∅
value Coordinator : C

AXIOM End-Exchange
∀X : XSituation |
    (∀D, S : C |
          D ∈ Detectors ∧ S ∈ StaticSupport∧
          Call(X.MdwOld, Coordinator, S, reqXEND, X.xid)∧
          Call(X.MdwOld, Coordinator, D, reqXEND, X.xid)) ⇒
        ⇔(∀D |
          D ∈ Detectors∧
          ReturnCall(X.MdwOld, Coordinator, D, reqXBEG, X.xid))

END
```

Étant donnée la conception d'architecture *middleware* donnée dans cette section, il est direct de vérifier que les contraintes relatives à la correction de l'échange de *middleware* sont satisfaites.

V Conclusion

En conclusion de ce document, nous proposons un résumé des contributions de notre travail, puis nous mettons en avant les perspectives faisant suite à notre proposition.

V.1 Contributions

Le développement de logiciel s'avère simplifié lorsque celui-ci est construit au-dessus d'une infrastrcuture *middleware*. Typiquement, une telle infrastructure gère les problèmes d'hérogénéité et d'interopérabilité, de manière transparente pour le développeur d'applications. De plus, les infrastructures *middleware* offrent en général des services qui fournissent des solutions aux problèmes fréquemment rencontrés lors de la construction de systèmes logiciels distribués complexes (*e.g.*, sécurité, transactions). Toutefois, la construction d'un *middleware* à partir de services fournis par une infrastructure n'est pas une tâche triviale. Cette tâche comprend :

- Le raffinement des exigences de l'application en une architecture concrète de *middleware* satisfaisant ces exigences,
- La sélection des implémentations des services appropriés composant l'architecture concrète,
- L'assemblage de ces implémentations en un *middleware* et finalement,
- La maintenance du *middleware* pendant l'exécution de l'application.

Le principal objectif de notre travail de thèse a été de proposer une méthode systématique pour la construction de *middleware* adaptés aux exigences des applications. Notre solution s'appuie sur la notion d'architecture logicielle. Précisément, nous nous appuyons sur un langage de description d'architectures pour la description précise de la structure et des propriétés d'un système logiciel, et pour concevoir des outils qui aident l'architecte, le concepteur et le développeur d'applications lors de la construction de *middleware* adaptés aux applications à partir de leur description architecturale.

Notre proposition inclut un système de stockage de *middleware* qui est utilisé pour mémoriser l'historique des divers raffinements d'architectures *middleware* réalisés par l'architecte. Ce système de stockage contient en outre des implémentations d'architectures concrètes de *middleware*. Le système de stockage de *middleware* s'accompagne d'une fonction de recherche efficace d'architectures *middleware* satisfaisant toutes, ou en partie, les exigences d'une application donnée. Cette fonction qui repose sur l'emploi du démonstrateur de théorèmes STeP, tout en en minimisant son utilisation dans un souci d'efficacité, est exploitée à la fois par l'architecte et le concepteur d'applications, le second l'utilisant pour trouver des implémentations correspondant à l'architecture *middleware* révélée par l'architecte. Un autre élément de notre solution est une méthode pour l'intégration d'implémentations d'architectures concrètes de *middleware*, pouvant être réutilisées par diffé rentes applications. En général, les langages de description d'architectures destinés à simplifier le processus de développement de logiciel, sont destinés à une infrastructure *middleware* particulière, propriétaire ou non. Notre contribution vient ici de la proposition d'une méthode générale d'intégration d'implémentations de *middleware*, qui est indépendante de toute infrastructure *middleware*.

Nous avons enfin introduit une méthode qui permet la construction d'architectures *middleware* pouvant être adaptées au regard des évolutions des exigences de l'application ou de l'infrastructure *middleware* sous-jacente. L'adaptation du *middleware* doit être réalisée en un temps fini et minimiser les perturbations en résultant pour l'exécution de l'application. De plus, l'adaptation du *middleware* ne doit pas affecter la cohérence de l'application. Dans le pire des cas où les exigences de l'application et l'infrastructure *middleware* changent de manière arbitraire, une architecture *middleware* en place doit être remplacée par une nouvelle architecture. Du point de vue architecturale, l'échange d'un *middleware* par un autre est équivalent à l'échange des connecteurs gérant les interactions entre les composants. Certains des langages de description d'architectures que nous avons examinés supportent le changement des composants de l'application. Toutefois, à notre connaissance, aucun n'aborde la modification des connecteurs d'une application à l'exécution. Notre proposition nous a conduit à définir un style architectural pouvant être utilisé pour la conception de toute architecture *middleware* dont l'échange peut être supporté tout en minimisant les effets en résultant pour l'exécution de l'application ainsi qu'en garantissant le maintien de la cohérence de celle-ci. Considérant la minimisation des perturbations pour l'exécution de l'application lors de l'échange du *middleware* sous-jacent, des travaux antérieurs dans le domaine de la gestion de la reconfiguration dynamique d'applications ont proposé d'identifier l'ensemble mimimal des composants de l'application affectés par la modification et de, au plus, perturber l'exécution de ces composants, les autres composants de l'application poursuivant leur exécution. Toutefois, lorsque l'on considère l'échange du *middleware*, tout composant interagissant avec le *middleware* est affecté par l'échange. La minimisation du nombre de composants affectés par l'échange du *middleware* n'offre donc aucun intérêt. Nous avons donc dû proposer une solution qui minimise la durée de l'échange, laquelle repose sur une détection fine du moment où l'échange du *middleware* peut effectivement survenir.

V.2 Perspectives

Au cours de notre présentation, nous avons supposé qu'un architecte obtient des exigences abstraites d'une application vis-à-vis du système d'exécution sous-jacent, et tente de les raffiner en une architecture concrète de *middleware* qui satisfait ces exigences. Nous avons alors proposé un système de stockage de *middleware* pour notamment mémoriser ces différentes étapes de raffinement, réalisées par l'architecte. Nous avons également introduit une fonction d'aide à la recherche d'architectures *middleware* au sein de ce système de stockage. Toutefois, discuté dans la référence [40], les exigences d'une application peuvent comprendre un ensemble de propriétés, éventuellement conflictuelles. Par exemple, une même application peut avoir des exigences relatives à des propriétés aussi variées que celles relevant de la sécurité, de la tolérance aux fautes, et des transactions. Il s'ensuit que plus d'un expert est nécessaire au raffinement des exigences d'une application en une architecture concrète de *middleware*, chacun des experts se concentrant sur le raffinement des exigences de son domaine. Il apparaît donc nécessaire de fournir un langage de description d'architectures qui permet de coordonner les activités réalisées par les différents experts. En d'autres termes, nous devons fournir le moyen de composer différentes vues architecturales d'un *middleware* et de vérifier que l'architecture en résultant satisfait certaines contraintes données.

Un point tout aussi intéressant dans le cadre de la définition de la relation de raffinement d'architectures *middleware* est de permettre l'identification systématique des parties d'une architecture *middleware* effectivement affecté suite à l'évolution des exigences de l'application. Ceci permettrait de proposer une méthode plus efficace pour le changement de *middleware* dans le cas d'une modification des exigences de l'application. Dans la solution que nous avons proposée, nous considérons un échange complet du *middleware* mais une modification plus localisée peut éventuellement être identifiée par l'architecte ou le concepteur de l'application. Une telle tâche peut néanmoins s'avérer complexe, et une méthode la simplifiant serait certainement utile. Une solution de base pourrait éventuellement s'appuyer sur le travail de la référence [59] qui définit une distance entre les spécifications de programmes.

Avec notre solution, le concepteur d'applications dispose d'un système de stockage d'implémentations de *middleware* et d'une fonction pour la recherche systématique d'implémentations disponibles d'éléments architecturaux, pouvant être réutilisées pour implanter une architecture concrète de *middleware*. Le concepteur doit alors évaluer si l'implémentation du *middleware*, résultante satisfait des critères quantitatifs comme, par exemple, l'extensibilité ou encore les performances à l'exécution. Les résultats de notre travail ne fournisse aucune aide pour cette évaluation du *middleware*, laquelle requiert d'utiliser des outils adéquats tant en termes d'évaluations théoriques qu'empiriques.

Configuration systématique de middleware

Systematic Customization of Middleware

I Introduction

Middleware is a software layer that stands between the operating system and the application. It provides reusable solutions to problems frequently encountered in many different types of applications, such as problems related to communication, reliability or security in distributed applications. Middleware hides details that have to do with heterogeneity and interoperability under a set of well-defined interfaces. Typically, middleware is not a monolithic block, but is built from middleware services, combined in a way that provides the functionality required by the application.

Developing middleware for a particular application comprises the following basic steps: (1) designing a concrete middleware architecture that satisfies requirements of an application; (2) implementing the middleware as prescribed by the concrete middleware architecture description; (3) maintaining the middleware so as to reflect changes in the application requirements and the availability of the middleware services. This thesis proposes a systematic method for the customization of middleware that meets requirements of an application. The proposed method aims at helping in all of the aforementioned steps that constitute the overall middleware development process. This chapter, further discusses the concept of middleware, then motivates the proposed approach for its systematic customization, and finally presents the field of software architecture which is the foundation of our proposal.

I.1 Middleware Infrastructures

The main features characterizing the successful development of software are identical to the features that generally characterize the successful accomplishment of every productive activity performed in todays life. The product should satisfy the requirements imposed by the consumers, and the tradeoff between the quality of the product and the time it takes to release it to the market should be successfully balanced. Part of the requirements imposed by the consumers are very much specific to the particular product while the rest of them is frequently met in many different other cases. Categorizing the various products regarding their requirements permits to form a hierarchy of product families. In the particular case of software products, distributed software applications constitute an interesting family because of the wide spectrum of common requirements imposed by this kind of applications and also because of the recognized difficulties to satisfy those

requirements. Employing well-known and generally approved solutions towards satisfying requirements that are common in different kinds of applications, allows to save precious time when developing a specific application. This time can then be devoted to satisfy requirements that are really specific to a particular application.

The previous remark led to the evolution of the middleware concept. Middleware is a software layer that stands between the operating system and the application [9] and provides well-known, reusable solutions to frequently encountered problems like heterogeneity, interoperability, security, dependability, atomic and isolated execution. Most of todays middleware infrastructures are based on the object-oriented paradigm and, thus, suggest an *object model* that should be followed for building applications and middleware using services provided by the infrastructure. Typically, *objects* are the main entities that make up an application or a middleware. Objects provide an interface that defines operations, which can be called to access and manipulate the internal state of the object. An *interface* is usually described in an interface definition language (IDL), that is specific to the infrastructure. Objects issuing a call are called *clients*, while objects receiving a call are called *servers*. An operation call can be divided into two parts: (1) a *request* for a service issued from a client object towards a server object; (2) a *response* from the server object that brings results back to the client object. A middleware infrastructure provides tools (e.g. IDL compilers) and a core software, often called a *broker*.

The broker provides functionalities used for mediating the interaction among objects. More specifically, the broker functionality enables forwarding requests and responses to objects [71], while handling interoperability and heterogeneity issues in a way transparent to the application. Application objects interact through a *proxy*, which combines functionalities offered by the broker. A proxy is divided in two parts, a *client-side proxy*, and a *server side proxy*. Then, interaction between objects takes place according to the following scenario:

- A client object calls an operation provided by the client-side proxy.
- The client-side proxy creates a corresponding request.
- The request is delivered to the server-side proxy.
- The server-side proxy transforms the incoming request into a call to the operation provided by the server object.

The purpose of the tools provided by an infrastructure, is to simplify the development of applications, which conform to the object model assumed by the infrastructure. More specifically, the tools usually generate proxies for client and server objects from the IDL description of the interfaces provided by the server objects. Finally, a middleware infrastructure provides a set of reusable services that comes along with the core software.

Lately, there have been attempts to come up with standards describing the semantics and the structure of middleware infrastructures, capable of supporting a wide range of

applications. The Open Distributed Processing Reference Model is such a standard (RM-ODP) [33]. Middleware infrastructures that comply to this standard should provide means for building applications that conform to the RM-ODP object model. According to that model, applications consist of objects having one, or more interfaces. An interface defines a set of possible interactions together with constraints on when those interactions may take place. RM-ODP compliant middleware infrastructures should further provide means to describe the overall architecture of the application in terms of rules that describe the application structure, and possible relations between the objects that make up the application. The RM-ODP standard does not require using any specific interface definition language to describe interfaces. Instead, it allows using different languages called *viewpoint languages* that enable to describe objects from different points of view (e.g. engineering, computational, enterprise, information etc.). Middleware infrastructures that comply to the RM-ODP standard should support a number of services, called *functions* [34]. More specifically, functions subdivide into the following categories:

- *Management functions*, for managing and monitoring the life-cycle of objects, threads, nodes, and clusters of objects. For instance, object management functions can be used to create, destroy, checkpoint objects, etc.
- *Coordination functions*, for coordinating objects, threads, clusters of objects. The coordination functions include among others, the event function for event based communication, the group function for grouping objects and guaranteeing certain properties for a group (e.g. guaranteeing some ordering in the interactions between objects), the transaction function for controlling and coordinating transactions.
- *Repository functions*, for storing, managing and trading information related to objects, interfaces, types, etc. For instance, the trading function mediates the advertisement and the discovery of interfaces.
- *Security functions*, for guaranteeing secure interaction among objects, secure management of information etc. Examples of such functions are the authentication, integrity, key management functions.

Standards like RM-ODP are quite abstract and do not give concise details on the behavior of functions. On the other hand, there exist standards like the Common Object Request Broker Architecture (CORBA), which are much more prescriptive regarding this kind of details (e.g. the CORBA object transaction service implements the well known two-phase commit protocol). The CORBA specification [69], is among the most successful attempts to standardize the semantics and the structure of middleware infrastructures. As usual, the CORBA standard defines an object model for building CORBA applications. CORBA compliant middleware infrastructures should provide support for building applications that conform to the aforementioned model. According to the CORBA object model, an application is a collection of objects. An object is an identifiable encapsulated entity that

may provide an interface defined in CORBA IDL. An interface is a set of operations that can be requested by objects. Requests are issued through a CORBA proxy, which combines functionality provided by the CORBA Object Request Broker (ORB). A CORBA proxy is divided into the client side proxy, called *stub*, and the server side proxy, called *skeleton*. CORBA compliant infrastructures provide a set of standard Common Object Services (COSs) [68] including:

- *Management services:* the LifeCycle COS for creating, destroying, transferring CORBA objects; the previous service together with the Relationships COS for managing clusters of objects.
- *Coordination services:* the Object Transaction Service (OTS) for controlling and coordinating transactions; the Concurrency Control Service (CCS), which provides primitive locking semantics; the event COS allowing for event-based communication and coordination, etc.
- *Repository services:* the naming COS for associating names to CORBA objects; the trading COSs used for registering, managing and trading CORBA objects.
- *Security services:* the security COS provides a variety of customizable authentication, authorization, and encryption policies.

Well known examples of CORBA compliant infrastructures include ORBIX [32], MICO [74], OMNIORB2 [47]. Usually, those infrastructures further provide additional functionalities that are not documented in the standard.

Except from standard middleware infrastructures, there exist others, which are developed either by well-known companies like Sun and Microsoft, or by research institutes. Sun, for instance, released the Enterprise JavaBeans (EJB) infrastructure, [85], for developing applications that comply to the EJB component model. This model basically extends the JavaBeans component model that was previously released by the same company [83]. According to the EJB model, an application consists of objects called beans. Objects can be either persistent or not. In the former case, the objects are called *entity beans* while in the latter case, objects are called *session beans*. Objects provide interfaces that export operations, which can be requested by clients. The JAVA RMI broker [84] is the basic communication platform that is used for accessing EJB objects. Hence, object interfaces are described using constructs provided by the JAVA language. As usual, a proxy that combines functionality of the JAVA RMI broker, is used to integrate JAVA RMI objects. EJB objects are hosted and managed by entities called EJB containers. The EJB infrastructure comes along with services that can be used to implement EJB containers. More specifically, the EJB infrastructure provides:

- *Management services:* EJB containers typically provide functionality for creating, destroying EJB objects, for managing threads, for allocating processes, etc.

- *Coordination services:* the JAVA Transaction Service (JTS) for coordinating transactions; the JAVA Messaging Service (JMS) for asynchronous communication.
- *Repository services:* the JAVA Naming and Directory Interface service (JNDI) for naming and directory services.
- *Security services:* for authentication and authorization control.

In the same spirit, Microsoft released DCOM, a middleware infrastructure based on the Distributed Component Object Model [58]. According to that model, an application consists of objects having one or more interfaces that can be requested by other objects. Interfaces are specified in MIDL, a DCOM-specific interface definition language. An interface contains operations that can be requested by client objects. Requests are issued through a proxy that combines functionality of the DCOM broker, which is based on ORPC. In principle, there is a proxy for each interface provided by a DCOM object. Except for the DCOM broker, the DCOM infrastructure includes:

- *Management services:* the DCOM broker provides a set of operations for creating and destroying DCOM objects; the Microsoft Transaction Server (MTS) [57] supports the same set of operations and it additionally provides *differed activation*, i.e. objects are created only when needed.
- *Coordination services:* the MTS service can be used for coordinating transactions; the Microsoft Message Queue service (MMQ) provides asynchronous communication semantics.
- *Repository services:* the DCOM broker provides a global name space that can be used to associate object classes, and object interfaces with global unique identifiers (GUIDs).
- *Security services:* the MTS includes a configurable security service.

Hence, using a middleware infrastructure instead of implementing an application from scratch allows the developer to save a great amount of time and effort while making at the same time his work clean and neat. However, the design of a middleware that meets requirements of an application also requires a significant amount of time and effort. From this standpoint, middleware infrastructures do not render any help, no matter how flexible and convenient they are.

I.2 Motivation

Summarizing so far, the development of distributed software gets much more simplified when building applications on top of a middleware infrastructure that deals with issues

like heterogeneity and interoperability in a way transparent to the application, and further provides reusable services solving problems that appear frequently in practice. More specifically, the developer is released from implementing protocols for interoperation, secure interaction among the entities that constitute the application, fault tolerance, atomic and isolated execution, and several others. However, the above tends to be slightly misleading since it introduces the advantages of implementing an application on top of a middleware infrastructure, while skipping the fact that the software development process implicates the design of a middleware architecture that satisfies requirements of an application. Typically, the design of a middleware architecture involves an architect and a designer. The input to the architect is the consumer's requirements, which are usually quite abstract, describing vague concepts. The role of the architect is to gradually refine those requirements into one, or more, concrete architectures (*i.e. architectures that could be implemented*) that satisfy the initial requirements [64]. Correctly refining abstract requirements into concrete architectures is a complicated task. Even in simple cases like the case study presented in [64], the refinement process requires time and effort from the architect.

The input to the designer is a concrete architecture handed to him by the architect. This architecture usually describes a configuration of components providing some properties but does not give implementation details. One of the designer's primary responsibilities is to check for available reusable software whose properties, when combined in the way prescribed by the architecture, satisfy the requirements of the application. In the specific case of a middleware architecture, the designer must look up for services provided by a middleware infrastructure, which can be combined in a way that satisfies the initial requirements. Attaining this goal also requires significant time and effort, especially when considering the large variety of reusable middleware solutions that are nowadays available. Take for instance the case where the requirements comprise those for reliable and isolated interaction. To satisfy those requirements, it is possible to use a CORBA compliant ORB together with a service that provides primitive locking semantics. Given the wide variety of ORB implementations and the variety of locking mechanisms that could be used, the designer work gets to be hard. Choosing, however, among the different options can be done by chance. In that case, it is the software developer who will, possibly pay the penalty. The input to the developer is a detailed description of the design decisions and his role is to assemble the middleware services selected by the designer into a middleware, according to the designer's specific guidelines. Most possibly, those guidelines prescribe the way to assemble the reusable software pieces, into an also reusable fraction of software. Getting back to the designer's decisions and how much they can either simplify, or complicate the developer's work, consider the case where among the different design options there exist services that support features making their integration much easier. Selecting those services instead of others is certainly more convenient for the developer. For example, some available ORBs support features like filters (ORBIX) or request interceptors (MICO), which allow to execute code just before, or right after a request is delivered to an object. Such a feature can be used to build a reusable code that acquires a lock on an object just before request delivery. Installing this code as a per-object-filter then alleviates the

developer's work who is not obliged to edit the source code of every individual object and to make the changes in-line. Another example is selecting a service that provides atomicity. Different implementations of the standard OTS service from CORBA exist. In most implementations, the application developer is supposed to explicitly provide source code for resource registration and resource recovery. However, in the OTS implementation coming from the ARJUNA framework [79], resource registration and recovery are implicitly performed by the service.

The design of a middleware gets even more complicated if the initial requirements include those for efficiency or reliability. The former imposes the need to evaluate the different design options in terms of performance. The requirement for reliability, obliges the designer to test the different options in terms of their exceptional behaviors, their resource limits, their dependability on hardware etc. Consequently, there is a need for rapid prototyping. It should be possible to assemble the middleware services and integrate the resulting middleware within the application as fast as possible. Finally, the requirements of an application or the availability of middleware services may evolve during the lifetime of the application. Such changes prompt the need to maintain a middleware so as to reflect the aforementioned changes.

Hence, developing a distributed software system still remains complicated and requires time and effort, even when building it, on top of available reusable solutions provided by todays middleware infrastructures. This, in fact, considerably reduces the benefits gained from reusing software. This thesis addresses the aforementioned issue by proposing a method for the *systematic customization of middleware which aims at helping the architect, the designer, and the developer, to design, develop and maintain a middleware that satisfies requirements of a particular application.*

I.3 Systematic Customization of Middleware

To customize means to build, fit or alter according to individual specifications [63]. Based on the remarks made in the previous section, building middleware amounts to designing a middleware architecture, out of a set of available middleware services, that meets the requirements of an application. Making middleware to fit amounts to integrating implementations of the individual architectural elements that constitute a concrete middleware architecture into a middleware, and integrating the result within the application. Altering middleware amounts to adapting a middleware architecture so as to reflect changes in the application requirements and the availability of middleware services. Hence, the basic objectives of this thesis are to provide a systematic method to assist the architect, the designer and the developer when designing, integrating, adapting middleware for a particular application. The overall approach is based on, and aims at continuing, the work that was previously proposed and published by previous and current members of the ASTER project, from the INRIA research institute, France. The innovations and contributions of this thesis are highlighted below.

The middleware design process can benefit from reusing available middleware architectures that satisfy the requirements of the application. Based on this concept, the first objective of this thesis is the design of a repository of available middleware architectures, that can be systematically traced by an architect for *retrieving* a concrete architecture that refines the abstract requirements handed by a consumer. Furthermore, the repository should be designed in a way that simplifies the work of the designer when trying to retrieve middleware services, whose properties, when combined in the way prescribed by a refined concrete middleware architecture, meet the initial requirements.

Previous work done in the context of the ASTER project is the basis to the proposed middleware design process. More specifically, the ASTER framework comprises a formal model for the specification of non-functional properties, provided by reusable software components, which are stored in a software repository [39]. The formal framework is based on an extension of first-order predicate logic. Non-functional properties can be either *abstract*, describing vague concepts (e.g dependability property, which states that the application is always in a correct state), or *concrete*, describing the behavior of a particular reusable mechanism (e.g. active replication). Moreover, a non-functional property P is said to refine another property P' if it is at least as strong as P'. Formally, $P \Rightarrow P'$. Based on the refinement relation the software repository is organized as a lattice structure of abstract properties, which are gradually refined into concrete properties. Given this well-organized repository, it is possible to efficiently retrieve software components that satisfy requirements of a particular application.

At this point, the author argues that this approach has a noticeable limitation. The proposed repository can be used by a designer to systematically retrieve middleware that satisfies abstract requirements of an application. On the other hand, such a repository is not very helpful for an architect. The existing refinement structure enables to trace changes across different refinement levels with respect to the strength of the properties provided by available middleware architectures, but it does not allow to exploit changes regarding the way an abstract architectural element is decomposed into an architecture of concrete architectural elements.

In principle, a middleware architecture, providing some properties, is made out of components that interact through a connector. Refining a middleware architecture that provides certain abstract properties into another middleware architecture that provides more concrete properties amounts to specifying precisely the structure and the properties of components and connectors that constitute the abstract architecture. Going through different refinement levels should reflect how abstract components and connectors are mapped onto concrete ones, or how abstract interfaces are mapped onto concrete ones, etc. Since the current ASTER repository contains only formal descriptions of properties and not structural descriptions of software, the mapping of the abstract architecture to the concrete one can not be traced by the architect. This information, however, is significant for describing a gradual refinement process that can be possibly reused. This thesis tries to cover the aforementioned limitation. Moreover, a method for navigating through the repository is

proposed. Previously, navigation through the repository structure was done with the use of a theorem prover. However, using theorem provers is a rather complicated task. In order to deal with the previous drawback, we propose a method that reduces the use of theorem proving technology, when searching for concrete middleware architectures that satisfy requirements of an application.

This thesis further proposes a systematic method for integrating a middleware implementation that can be easily reused within a given application. This method, together with an adequate code generation tool, simplifies the work of the developer, and further enables fast prototyping, which makes the performance evaluation of different design options less costly. Previous work done in the context of the ASTER project includes a code generator that automatically builds a prototype application on top of the ORBIX middleware infrastructure [39]. Hence, support for integrating a middleware architecture within a given application was rather primitive. Based on this fact, the second objective of this thesis is to propose an approach and an adequate tool for integrating a middleware within a particular application, which does not depend on the underlying middleware infrastructure. From now on, the term *middleware synthesis* is used by the author to refer to both the processes of retrieving a middleware architecture and integrating it within a given application.

Adapting a middleware in the general case where both the requirements of the application and the availability of middleware change arbitrary, requires performing major structural changes to the architecture of the middleware. Thus, adapting a middleware architecture, in the worst case, requires exchanging it with a new one. Middleware adaptation should introduce minimal disruption in the execution of the application and preserve application consistency. Moreover, middleware adaptation should take place in a finite time. Based on those remarks, the third objective of this thesis is to suggest an architectural style for designing concrete middleware architectures that can be exchanged while preserving the aforementioned requirements. So far, the issue of dynamically changing the middleware was not addressed in the ASTER framework.

Designing, developing, and maintaining middleware systematically requires a precise and unambiguous way to describe the structure and the properties that characterize the behavior of middleware. In the current practice, however, the structure and the properties of middleware are specified in a semi-formal manner. Middleware services are described in terms of interfaces, specified using an infrastructure-specific IDL. The behavior of middleware services is given in a natural language. This lack of precision renders the systematic customization of middleware virtually impossible. Middleware architecture refinement requires a tool that verifies correctness of the individual refinement steps. This tool should be able to reason about whether the properties that characterize the behavior of a concrete architecture refine the properties that characterize the behavior of an abstract architecture. If those properties are expressed in a natural language it would not be feasible to devise such a tool. Hence, some sort of formal description of the middleware behavior is needed towards the systematic customization of middleware. Systematic integration of a middleware implementation, out of existing middleware services requires to precisely describe how

those fractions of software are structured.

The software architecture community makes a remarkable effort towards formalizing the basic principles that govern the art of building software [70]. Describing formally the basic architectural elements that constitute a software, their behavior, and the way they are structured into a whole allows to systematize several parts of the overall process of building software. For instance, it becomes possible to verify that the overall system complies with a standard [48]. Another example is to verify the correct behavior of the system based on the behaviors of the individual architectural elements [51] or to reason about the requirements of the system and build it out of available elements that satisfy those requirements [76]. Finally another example is the evaluation of a software system based on well defined criteria like modifiability, scalability etc. [41]. In the context of this thesis, the software architecture paradigm is followed towards the precise and unambiguous specification of both the structure, and the behavior of middleware.

I.4 Document Structure

This section presents the structure of the rest of this document. Chapter 2 briefly presents the foundation of the software architecture paradigm. More specifically, it presents Architecture Description Languages (ADLs), i.e. languages that provide means to describe the structure and to analyze the properties of software. Moreover, this chapter proposes an architecture description language (ADL) for the systematic customization of middleware. The proposed ADL is evaluated throughout the rest of the document, based on the evaluation framework proposed in [56]. Chapter 3 details the systematic middleware synthesis process. More specifically, it details the design of a repository of middleware architectures. The repository structure reflects possible refinement relations among available middleware architectures. Moreover, Chapter 3 details the systematic integration of middleware. Chapter 4 discusses the basic requirements for adapting a middleware architecture while minimizing the disruption introduced in the execution of the application that lies on top of the middleware, and while preserving the application consistency. Moreover, this chapter discusses the requirements on software for building middleware that can be adapted. Chapter 5 concludes this document with a brief summary, the contribution of this thesis, the current status and the future perspectives of this work.

II Architecture Specification

In the current practice, both the behavior and the structure of middleware are specified in a semi-formal manner. More specifically, a middleware comes along with an interface description, given in an interface definition language specific to the middleware infrastructure, and a description of its structural and behavioral properties, which are delineated using natural language expressions. On the other hand, building software systematically requires a precise and unambiguous way to describe the properties and the overall structure of software. To satisfy this demand, the software architecture community proposed Architecture Description Languages (ADLs) [17, 35, 56, 36], i.e. languages that provide means to formally describe the structure and the properties that characterize the behavior of software.

This chapter gives an overview of the basic features provided by architecture description languages and proposes using an ADL for the systematic customization of middleware. The ADL provides means to describe components that make up a software system and connectors that mediate the interaction between components. The suggested ADL comes along with a formal model for describing the properties that characterize the behavior of software. This model is based on linear temporal logic, but can also be realized on top of other existing formalisms.

II.1 Architecture Description Languages

Existing ADLs proposed in the context of different architecture-based development and analysis environments (ASTER[39], CONIC [52], C2 [87], DARWIN [50], DCL [6], DURRA [7], RAPIDE [49], SADL [64], UNICON [78], WRIGHT [2]) agree in that the starting point in an ADL specification should be to describe the basic *components* that make up a software application. The next step in the specification should be to describe the architectural elements that model the interaction among the components. The latter are usually called *connectors*. Finally, an ADL specification should describe how components and connectors are interconnected to form the application, this description is usually named *configuration*.

However, there are certain differences in the way existing ADLs describe the basic architectural elements mentioned above. Differences may be either syntactic, or semantic. In

the former case, different syntax is adopted towards specifying an architecture. In the latter case, the basic architectural elements are characterized by different features. Furthermore, there exist also qualitative differences among the different ADLs. Typically, qualitative differences originate from the way connectors are viewed in existing ADLs. For instance, depending of the ADL, a connector is, or is not considered as a first class architectural element. Another example is that some ADLs support only a limited number of different connectors or even a single connector, usually called a bus [73]. In [56], this point is used to distinguish between ADLs and Module Interconnection Languages (MILs). The remainder of this section gives an overview on the basic features characterizing components, connectors and configurations, and further points out similarities and differences among existing ADLs, regarding those basic features.

II.1.1 Basic features characterizing a component

A *component* is a unit of data or computation and the basic features that characterize it are its interface, type and properties.

A component *interface* describes a number of interaction points between the component and the rest of the architecture. Most ADLs examined in the scope of this thesis support this particular feature. However, several syntactic and semantic differences have been observed between them. In ASTER, for instance, components export interfaces to the environment and import interfaces from other architectural elements. An interface defines a set of operations. The syntax follows the one proposed by OMG in the CORBA IDL standard [69]. In CONIC, an interface describes a set of entry and exit ports which are typed (e.g. events). In DARWIN, CONIC's successor, an interface describes services required from and provided by a component. In DCL, components are called *modules*. A module is a group of actors, i.e a group of processing elements that communicate through asynchronous point-to-point message passing [1]. A module description comprises a set request rules which prescribe the module interface. A component interface in C2 defines two basic points of interaction named top and bottom ports. These ports are used by a particular component to accept requests from, and issue requests to, components that reside either above, or below it (the architecture is topologically structured). A component interface in UNICON defines a number of interaction points, called players. Players are typed entities. The type of a player can be out of a limited set of predefined types. In WRIGHT, a component interface defines input and output ports. Pretty similar is the way interaction points are defined in DURRA. In RAPIDE, the points of interaction can be either services required from or provided by a component, or events generated by a component. Finally, in SADL, an interface is just a point of interaction.

A component *type* is a template used to instantiate different component instances into a running configuration. All of the ADLs examined in this paper distinguish between component types and instances. Types are usually extensible. Sub-typing (e.g. in C2, ASTER) is a typical method used to define type extensions. In DARWIN and RAPIDE, types are

extended through parameterization. UNICON, however, only assumes a set of predefined types.

Component *properties* define the component's observable behavior. In WRIGHT behavior is described in CSP [29, 30]. In RAPIDE, partially ordered sets of events (posets) are used to describe component behavior. In the very first version of DARWIN, properties were described in CCS [61]; in the latest version properties are described in pi-calculus which extends the semantics of CCS with means that allow to describe the dynamic instantiation of processes [60]. In DCL, the behavior of a module is deduced by the behaviors of the actors that constitute the module. An extension of the basic ACTORS formalism is used to describe the behavior of actors [5] within a software architecture. C2 defines causal relations between input and output points. Finally, in the first version of ASTER, first-order logic was used to describe properties, which was then extended with an operator that defines a precedence order in the verification of predicates. Similarly, in SADL, Temporal Logic of Actions TLA [45] is used to describe component properties.

II.1.2 Basic features characterizing a connector

A connector is an architectural element that models the interaction among components. Its basic features are again its interface, type, and properties.

Typically, the basic features that characterize connectors are described in a way similar to the one used for components. In WRIGHT and UNICON, for instance, the interface of a connector is a set of interaction points, named roles. In DURRA, a connector is called channel and its interface is defined in the very same way as a component interface. In C2, and SADL connector interfaces are described using the same syntax as the one used to describe component interfaces. In DCL, connectors are again groups of actors, called *protocols*. Protocols define a set of roles describing the way interaction takes place among modules. Except for the case of UNICON, connector types are extensible. As expected, connector properties are described using the same formalism as for the component properties.

Unlike components, connectors are not always considered as basic architectural elements (e.g. CONIC, DARWIN, RAPIDE). It is worth noticing that in the case of ASTER, the ultimate goal is to locate a middleware architecture that mediates the interaction between application components in a way that satisfies their requirements. Hence, connectors are not defined explicitly in the architectural description of an application, instead they appear as bindings between components. Despite this fact, connectors *are*, conceptually, considered as basic architectural elements.

II.1.3 Basic features characterizing a configuration

A configuration (or topology, structure) is a relative arrangement of component and connector instances. A configuration is, typically, described in terms of bindings between

points of interaction. Several ADLs either assume or provide means to describe *constraints* on the bindings, the behavior, and the evolution of a particular configuration. Those constraints together with constraints that determine the correct use of the rest of the basic architectural elements (components and connectors) form what is called an *architectural style*. In particular, Moriconi et al. in [64] define architectural style as:

"... a vocabulary of design elements, well-formedness constraints that determine how they can be used, and a semantic definition of the connectors associated with the style."

Constraints may simply describe restrictions on the way components are bounded. In DARWIN, for instance, only bindings between required and provided services are allowed. In ASTER, the types of bound interface instances should match. Moreover, constraints may be on the behavior of the overall configuration. In ASTER, for example, requirements for security and dependability, can be specified for a particular configuration. In DCL, actors can interact with each other only if they belong to the same module. Actors external to a module can interact with ones internal to the module only if they were previously admitted by the module itself. RAPIDE also allows to describe constraints on the behavior of a particular configuration. Constraints may also relate to the (dynamic) evolution of a particular configuration. In DURRA and RAPIDE, for example, it is possible to describe conditions under which a configuration changes into another one.

II.1.4 ADL evaluation framework

An interesting framework for evaluating different ADLs is proposed in [56]. According to that framework, an ADL should provide means to easily describe the basic architectural elements in a simple and understandable syntax. Hence, it should be *expressive* and *understandable*.

Moreover, an ADL should provide means for describing software in different levels of detail. It should be possible, for instance, to abstract away several details within composite components and connectors made out of more concrete components and connectors. It should also be possible to reveal several details, by looking inside abstract composite components and connectors made out of more concrete components and connectors. Hence, an ADL should support *compositionality* or *hierarchical composition*.

An ADL should enable the correct and consistent *refinement* of architectures into concrete executable systems. Furthermore, an ADL should enable the *traceability* of changes across different refinement levels. An ADL should provide means to chain a sequence of correct architectures in an hierarchy, allowing the designer to observe changes among them, to conclude that the most concrete architecture is correct with respect to the most abstract one, and to detect inconsistencies in the horizontal composition of architectures.

Scalability is another important issue. An ADL should provide support that allows to describe, analyze, develop, and maintain large-scale real world applications. Finally, an ADL should provide support for component, connector and configuration *evolution* and *dynamic*

evolution. It should be possible to add, remove, change components and connectors that make up an application, while introducing minimal disruption in the execution of the application.

Those main points used in [54] for the evaluation of an architecture description language entirely conform with the main objectives introduced by the systematic customization of middleware. More precisely, an ADL used for the systematic customization of middleware should allow to easily and accurately describe complex applications that may possibly grow in size. Moreover, the ADL should allow to easily and accurately describe the structure and properties of a middleware architecture. The ADL should further provide support for refining abstract application requirements into a concrete middleware architecture. An ADL should provide support that enables to easily integrate the implementation of a concrete middleware architecture within an application resulting into a complete executable software system. Finally, an ADL should address issues that relate to the consistent adaptation of a middleware architecture. Among the ADLs examined in this document, ASTER tries to address the previous issues for the specific case of middleware. For that reason, it becomes the basis of the work presented in this document, which further suggests methods that complete and enhance the ones previously proposed for the systematic customization of middleware.

II.2 ADL for the Systematic Customization of Middleware

Based on the software architecture paradigm and towards the systematic customization of middleware that satisfies requirements of an application, this thesis proposes using an ADL, as the vehicle for precisely describing the structure and the properties of middleware. The proposed ADL provides means to describe the structure and properties of *components*, which can be either of the application, or middleware services, and *connectors* made out of middleware services. The remainder of this section gives more details regarding the specification of those elements. The exact syntax of the ADL used is an enhanced version of the syntax traditionally adopted by ASTER and for that reason the ADL grammar is not much detailed in the remainder of the document. In particular, the basic enhancement is that connectors are explicitly defined within an ADL description, for the better presentation of an architecture.

II.2.1 Describing components

The description of components comprises the following basic features: a *type*, which defines a set of *interfaces*, required from, or provided by a component. A component type is used to create multiple *component instances*. To enable describing components in different levels of detail, a component type also allows defining an instance of a *configuration*

that makes up a component, i.e. an instance of a set of components connected through a connector. In case where the description of a component type defines a configuration, the component is said to be *composite*. In the opposite case, the component is said to be *primitive*. An *interface* is of a specific *interface type*, which describes a set of operations. The syntax used to describe interface types is very similar to the ones proposed by the IDLs supported by todays widely known middleware infrastructures (e.g. CORBA IDL compliant infrastructures, JAVA language - interface constructs, DCOM MIDL etc.).

Finally, a component type allows describing component properties. Properties of a component describe the component's observable behavior as it interacts with other architectural elements. The specification of properties is discussed in more detail at the end of this section. The following example shows the notation used throughout this thesis to describe middleware and application components. In particular, the described component is a composite one, made out of a configuration `hc` of type `HorizontalComposition`.

```
component Composite {
   requires // list of required interfaces;
     InterfaceTypeA InterfaceInstanceA;
   provides // list of provided interfaces;
     InterfaceTypeB InterfaceInstanceB;
   configuration HorizontalComposition hc;
   properties       // component's observable behavior

};
interface InterfaceTypeA {
   // list of required operations;
   RetType op(in argType arg, out argType' arg', ...);
};
interface InterfaceTypeB {
    ....;
};
```

II.2.2 Describing connectors

The purpose of the systematic customization of middleware is to design and implement a connector that represents a middleware architecture, which mediates the interaction among application components in a way that satisfies application requirements. Hence, a connector defines types of architectural elements that constitute the middleware architecture. The basic architectural elements that constitute a middleware architecture can be components, more primitive connectors and a configuration that combines instances of components through instances of more primitive connectors. A connector has a *type*,

which is used to define different connector instances within an ADL description. Defining a connector instance is equivalent to defining an instance of the configuration described in the connector's architectural description. Finally, a connector's description comprises properties provided by a connector. Connector properties characterize the interaction among components that are bound through an instance of this connector. For example, connector properties may state that the interaction among components bound through an instance of this connector is reliable and synchronous.

According to Section I.1, a *concrete* middleware connector that can be built on top of todays middleware infrastructures, represents a middleware architecture, which defines at least component types that correspond to client-side and server-side proxies. Client-side and server-side proxies combine functionality provided by a broker in a way that allows application components to interact. Moreover, a concrete connector defines a configuration type which comprises component instances, that represent client-side and server-side proxies, bound through a more primitive connector (e.g. a primitive RPC connector, or a primitive TCP/IP connector). Except for the client-side and server-side proxies, a concrete connector may define component types that represent other middleware services provided by a middleware infrastructure. Then, the connector's configuration may comprise instances of components that represent middleware services. Functionality provided by middleware services is combined together with functionality provided by the broker, in a way that allows application components to interact.

However, given that the main goal of the systematic middleware synthesis is to design and implement a concrete connector that satisfies application requirements, the ADL description of an application does not need to describe in detail the elements and the structure of a connector. Hence, several of the previous features used to describe a connector can be omitted when describing the structure and the requirements of an application. More precisely, the only field that has to be specified within the ADL description of an application is the properties guaranteed by the connector. The following fraction of code is an example of such a connector description.

```
connector MdwConn {
   properties
      // e.g reliable communication
};
```

II.2.3 Describing configurations

The description of a configuration comprises the following basic features: a *type* which defines a set of component instances interconnected in terms of bindings among required and provided interfaces, realized through a connector instance. A configuration type is used

within an ADL description to define one, or more instances of a configuration. The following is an example of the notation used to describe a configuration. More specifically, the following fraction of code describes a configuration type, named `HorizontalComposition`. This type defines two component instances of types `ConstituentC` and `ConstituentS` interconnected through a connector of type `MdwConn`.

```
configuration HorizontalComposition {
    instances // list of component/connector instances;
     ConstituentC C;
     ConstituentS S;
     MdwConn MdwConnInst;
    bindings // list of bindings between required and provided interfaces ;
     C.Interface to S.Interface through MdwConnInst;
   };
```

II.2.4 Specifying middleware properties

As mentioned at the beginning of this chapter, middleware specification is semi-formal in that the structure and the properties of middleware are delineated using natural language expressions. The proposed ADL allows, so far, to describe formally the structure of software. What is left to discuss is the formal specification of its properties.

Choosing among available formalisms

A variety of different formalisms are being used for specifying the behavior of software. Among the prominent ones, we meet process algebras and logics.

Process algebras provide structured methods for the analysis of discrete event systems and can be further divided into three different styles [31]. The *Denotational* style, is the basis in the theory of Communicating Sequential Processes (CSP) introduced in [30], and relates a software to a specification of its observable behavior. The *algebraic* style, is the foundation in the definition of the Algebra for Concurrent Processes (ACP) [8]. Mainly, it provides equations and in-equations for comparison, transformation and optimization of software. The *operational* style, that was used to define the Calculus of Communicating Systems (CCS) [61], describes individual steps of a possible software implementation.

Logics can be divided into temporal logics, which support operators that allow to reason about the temporal relation of the predicates that constitute a formula, and logics in which time can only be modeled as a quantitative parameter. Linear time temporal logics and branching time temporal logics are the two basic variations of temporal logics [20]. According to the former, time is linearly modeled, meaning that at a given moment only one future is possible. According to the latter, time has a branching, tree like, nature. This

means that at a given moment, time may split into alternate paths representing different possible futures.

Choosing a formalism for the systematic customization of middleware is more or less subjective since the borders that divide available formalisms into categories regarding their suitability for specific uses are not, and possibly cannot be, precisely defined.

The basic objective is, however, to choose a formalism that allows to specify middleware properties in various levels of abstraction. The reason behind this is summarized in the following points:

- First, to allow consumers to describe rather abstract and vague concepts, which is, in fact, the typical case when describing requirements on a product. For instance, the chosen formalism should allow to describe very abstract properties like dependability, i.e. despite failures a software system progresses and reaches a correct state.
- Second, to allow an architect to refine those abstract requirements into properties provided by a concrete middleware architecture that could be implemented. For instance, the chosen formalism should enable the refinement of the dependability property into passive or active replication.
- Third, to allow a designer to reason about properties provided by specific mechanisms incorporated within a middleware architecture. For example, the chosen formalism should allow to describe the behavior of a middleware service that provides replication i.e. takes as input a request sent by a client component, and outputs replicas of this request towards a group of identical server components.

As stated in [4], logics can be used to express more abstract properties than input-output behavior. Indeed, there exist various examples that verify this statement. In [76], for instance, predicate logic extended with a precedence operator was used to express dependability properties in various levels of abstraction. Moreover, in [38], first order predicate logic was sufficient for expressing abstract interaction properties. Finally, temporal logic was used in [45] to describe lower level input-output behavior. Based on those remarks, the first design step was to chose logic as a formalism for describing the behavior of middleware.

Choosing between temporal logics and logics that do not give means to describe temporal relations among the predicates that constitute a formula is the next design step. By observing the evolution of ASTER, it is easy to notice that first order predicate logic might be convenient to express simple interaction properties [38], and security properties [10], but when getting to cases like dependability [39], and transactions [92] it is clear that temporal operators become indispensable.

Finally, choosing between linear and branching temporal logic is not difficult. Branching temporal logic is ideal for model checking because of the existence of a very efficient algorithm for finite state programs. It is also very handy when there is a need to query

for a property along some execution path of a software system. On the other hand, linear temporal logic is used to describe properties that hold over all execution paths [20]. In the case of the systematic customization of middleware, a middleware is required to guarantee specific properties over the execution of the application (e.g. reliable communication, atomic and isolated execution of a set of requests), making thus linear temporal logic a sufficient vehicle. Finally, note that depending on the properties that need to be specified, there might be cases where linear temporal logic should be extended. Typical examples are the real time temporal logics and probabilistic logics used for describing properties of real-time and multi-media systems [15].

Summarizing so far, linear temporal logic is going to be used for describing middleware properties. More specifically, the remainder of this section introduces some basic temporal logic notation used hereafter in the specifications. The remainder of this section further presents a general specification model, which defines, based on the chosen formalism, the basic concepts introduced by the software architecture paradigm (e.g. component, interface, configuration etc). The same model could be defined on top of another formalism like the ones previously mentioned. A interesting remark that could be made at this point, is that logics are flexible in that they allow to define new constructs like components, interfaces etc. On the other hand, process algebras provide fixed constructs. Hence, if we were to use a process algebra to describe middleware properties, we would have to map the basic concepts of software architecture into concepts introduced by the algebra (e.g. map components to processes). This mapping is sometimes straight-forward, but there are also cases where it is not that obvious (e.g. dynamic component instantiation could not be expressed using Ccs; this led Darwin to adopt the pi-calculus for describing software properties). In conclusion, the author believes that using temporal logic instead of a process algebra, simplifies the definition of a flexible model for the specification of middleware properties.

Temporal logic notation

The basic temporal logic notation used hereafter to describe middleware properties is introduced in [53]. More specifically:

- Symbols $\wedge$, $\vee$, $\neg$, $\Rightarrow$, denote the logical *and*, *or*, *not*, and *implication* respectively.
- Symbols $\forall$, $\exists$, are used to denote the universal and existential quantifiers respectively.
- Symbols $\oplus$, $\ominus$ denote respectively the *next*, *previous* temporal logic operators. $\oplus P$, resp. $\ominus P$, states that P holds at the next moment in time, resp. that P held at the previous moment in time.
- Symbols $\Diamond$, $\Diamondblack$ denote respectively the *eventually* and *once* temporal logic operators. $\Diamond P$, resp. $\Diamondblack P$, states that P holds at some time in the future, resp. that P held at some time in the past.

- Operators $\Box$, $\boxminus$, denote respectively the *henceforth*, and the *has always been* temporal logic operators. $\Box P$, resp. $\boxminus P$, state that P holds from this time on, resp. that P held until this time.

Moreover, the properties characterizing a component or a connector are given as axioms that make up a theory. The syntax used, hereafter, to define such a theory is based on the one suggested by the vendors of the Stanford Temporal Logic Theorem Prover (STeP) [1]. STeP, supports basic types like integer (denoted by *int*), boolean (denoted by *bool*), real (denoted by *real*), and allows for user-defined types like arrays (defined as `array` [*type*] `of` *type*), tuples (defined as $(type, type', \ldots, type'')$), records (defined as $\{id : type, id' : type', \ldots id'' : type''\}$) etc. In addition, $X[i]$ denotes the *ith* element within an array X; $\sharp(i)t$ denotes the *ith* element within a tuple t; $\sharp(id)t$, denote the element *id* within a record t [2]. For more information about temporal logic specifications in STeP, the reader can refer to [12].

Specification model

In order to describe in temporal logic, middleware properties required by the application, or provided by a middleware, there is a need to devise a basic formal specification model, which consists of:

- A vocabulary of basic types used to describe basic architectural elements and features that recursively characterize those elements. e.g. components, configurations, interfaces, requests, operations, failures etc.
- A number of axioms that describe the correct use of elements belonging to the vocabulary.

According to Moriconi *et. al.* [64], such a formal specification model, together with a set of axioms describing properties guaranteed by a number of connectors, defines *an architectural style*. Still following the terminology used in [64], the specification model used to describe middleware properties is called, hereafter, an *architectural style theory* for the specification of middleware properties. From a practical point of view, the architectural style theory for the specification of middleware properties is defined as a STeP theory called `Base-Architectural-Style`. The basic types of elements defined in the `Base-Architectural-Style` theory are the following.

- `type` $\mathcal{R}$, which is used in the specifications to define elements of type request. Then a request is defined in a specification as a value of type $\mathcal{R}$.

[1] Note that in some cases the STeP syntax is simplified for the better presentation of this document.
[2] For simplicity, in most cases *t.id* is used to refer to fields of a record.

$$value\ req : \mathcal{R}$$

- `type` $\mathcal{UID}$, which is used in the specifications to define unique identifiers which, in turn, are used to distinguish among identical values. As a typical example, requests are not unique, and hence, unique identifiers are used to distinguish between them.

- `type` $\mathcal{C}$, is used in the specifications to define elements of type component, i.e. component instances. More specifically, component type is defined to be a record that has three fields:

  $$\texttt{type}\ \ \mathcal{C} = \{I_p, I_r, \Sigma_C\},$$

  Where $I_p = \{i : \mathcal{I}_p\}$ is a set of provided interfaces, $I_r = \{i : \mathcal{I}_r\}$ is a set of required interfaces, $\Sigma_C = \{\sigma : \Sigma\}$ is a set of possible states; corresponding types are defined below. Hence, a component instance C in the specification of a property is a value of type $\mathcal{C}$:

  $$value\ C : \mathcal{C}$$

- `type` Σ, which is used in the specifications to define elements of type state i.e. component states:

  $$\sigma : \Sigma$$

- `type` $\mathcal{I}_r$ is used in the specifications to define required interface instances. More specifically, $\mathcal{I}_r$ is defined to be a set of required operations:

  $$\texttt{type}\ \ \mathcal{I}_r = \{o_{C_i} : \mathcal{O}_r\},$$

- `type` $\mathcal{O}_r$, is used in the specifications to define required operations. More specifically, the required operation type is defined to be a function that takes as parameters the state of a calling component and a unique identifier, and returns a request that corresponds to the call.

  $$\texttt{type}\ \ \mathcal{O}_r : \Sigma * \mathcal{UID} \rightarrow \mathcal{R}$$

  Then, a required operation o is an instance of the above type, i.e. $value\ o : \mathcal{O}_r$, and $rang(o)$ denotes the range of o, i.e. the set of requests o returns as an outcome.

  Moreover, the following axiom is assumed to hold for the operations of a required interface:

$$\begin{array}{l}\texttt{AXIOM Require-Unique-Signatures}\\ \quad \forall I : \mathcal{I}_r,\\ \quad o_i, o_j : \mathcal{O}_r \mid \\ \quad ((o_i \in I) \wedge (o_j \in I) \wedge (o_i \neq o_j)) \Rightarrow rang(o_i) \cap rang(o_j) = \emptyset\end{array}$$

- `type` $\mathcal{I}_p$ is used in the specifications to define provided interface instances. More specifically, $\mathcal{I}_p$ is defined to be a set of provided operations:

$$\texttt{type}\ \ \mathcal{I}_p = \{o_{C_i} : \mathcal{O}_p\},$$

- `type` $\mathcal{O}_p$, is used in the specifications to define provided operations. More specifically, the provided operation type is defined to be a function that takes as parameters the state of a receiving component, a request, and a unique identifier, and returns the state of the receiving component after the request is served.

$$\texttt{type}\ \ \mathcal{O}_p : \Sigma * \mathcal{R} * \mathcal{UID} \rightarrow \Sigma$$

  Then, a provided operation o is an instance of this type, i.e. $value\ o : \mathcal{O}_p$, and $dom(o)$ denotes the domain of o, i.e. the set of tuples o accepts as parameters.

  Finally, the following axiom is assumed to hold for the operations of a provided interface:

$$\begin{array}{l}\texttt{AXIOM Provide-Unique-Signatures}\\ \quad \forall I : \mathcal{I}_p,\\ \quad o_i, o_j : \mathcal{O}_p \mid \\ \quad ((o_i \in I) \wedge (o_j \in I) \wedge (o_i \neq o_j)) \Rightarrow dom(o_i) \cap dom(o_j) = \emptyset\end{array}$$

- The symbol $\rightsquigarrow$ is used in a specification to denote a function that takes as parameters two interface instances and returns true if those instances are interconnected and false otherwise:

$$\rightsquigarrow : \mathcal{I}_r * \mathcal{I}_p \rightarrow \{true, false\}$$

  Moreover, the following axiom holds:

$$\begin{array}{l}\texttt{AXIOM Type-Checking}\\ \forall i : \mathcal{I}_r,\\ j : \mathcal{I}_p \mid \\ (i \rightsquigarrow j) \Rightarrow \\ (\forall o_r : \mathcal{O}_r \mid \\ (o_r \in i) \wedge \\ (\exists o_p : \mathcal{O}_p \mid (o_p \in j) \wedge (rang(o_r) = \sharp(2) dom(o_p)))\end{array}$$

- $\mathcal{CONF}$ is used in the specifications to denote a configuration. More specifically the configuration type is defined to be a set of component instances bound together:

$$\begin{array}{ll}\texttt{type} & \mathcal{CONF} = \{C : \mathcal{C} \mid \\ & (\exists C' : \mathcal{C} \mid (C' \in \mathcal{CONF}) \wedge \\ & (\exists i, i' \mid (i \in \sharp(I_r)C \vee i \in \sharp(I_p)C) \wedge (i' \in \sharp(I_r)C' \vee i \in \sharp(I_p)C') \wedge \\ & ((i \rightsquigarrow i') \vee (i' \rightsquigarrow i))))\}\end{array}$$

- The symbol [ ] is used in a specification to denote a function that takes as argument a set of component instances and a state, and returns true if the set of components is currently in this state:

$$[\,] : \{C : \mathcal{C}\} * \Sigma \rightarrow \{true, false\}$$

- $response() : \mathcal{O}_r$ is an instance of a required operation that takes as input a unique identifier $rid : \mathcal{UID}$ and the state of a receiving component $C : \mathcal{C}$ and returns a response $resp : \mathcal{R}$ to the request associated with rid.

- $response() : \mathcal{O}_p$ is an instance of a provided operation that takes as input the state of a calling component $C : \mathcal{C}$, a response $resp : \mathcal{R}$ and the unique identifier $rid : \mathcal{UID}$ associated with it, and returns the state of the calling component after receiving the response.

- The symbol $failure()$ is used in a specification to denote a function that takes as argument a request and returns true if the request indicates a system exception, and false otherwise.

$$failure : \mathcal{R} \rightarrow \{true, false\}$$

Based on the previous definitions it is possible to define the following macros that are used in the specifications to prescribe interaction among components of a configuration:

- The following macro holds for two component instances, C, C', if those instances are elements of a configuration $Conf$, and a request req, is a result of a call to operation o that belongs to an interface ir required by C, which is bound on a interface ip provided by C'.

$$
\begin{array}{l}
\texttt{macro} \\
\quad Call(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad \exists o : \mathcal{O}_r, \\
\qquad ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad ((ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in ir) \wedge \\
\qquad\quad (req \in rang(o)) \wedge \ominus[C, \sigma] \wedge req = o(\sigma, rid))
\end{array}
$$

- The following macro holds for two component instances, C, C', if those instances are elements of a configuration $Conf$, and C' is in a state, which results from a call to an operation o that belongs to an interface ip provided by C', which is bound on a interface ir required by C.

$$
\begin{array}{l}
\texttt{macro} \\
\quad UpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad (\exists ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad o : \mathcal{O}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad (ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in \sharp(I_p)C') \wedge \\
\qquad\quad ((\sigma, req, rid) \in dom(o)) \wedge \\
\qquad\quad \Diamond Call(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge [C', o(\sigma, req, rid)]
\end{array}
$$

- The following macro holds for two component instances, C, C', if those instances are elements of a configuration $Conf$, and C' returns a response $resp$ to a request req made by a component C.

$$
\begin{array}{l}
\texttt{macro} \\
\quad ReturnUpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad \exists req : \mathcal{R}, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad \Diamond UpCall(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge resp = response(\sigma, rid)
\end{array}
$$

- The following macro holds for two component instances, C, C', if those instances are elements of a configuration $Conf$; and C is in a state that results after receiving a response $resp$ to a request req issued to C'.

$$\begin{array}{l}\texttt{macro}\\ \quad ReturnCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) =\\ \qquad \exists \sigma : \Sigma \mid\\ \qquad\quad \Diamond\!\!\!-\, ReturnUpCall(Conf, C, C', resp, rid) \wedge\\ \qquad\qquad \ominus [C, \sigma] \wedge [C, response(\sigma, resp, rid)]\end{array}$$

Based on the previous architectural style theory it is possible to describe basic interaction properties (e.g. reliable RPC communication) provided by a middleware architecture. Moreover, with simple extensions of the previous style, it is possible to describe more complex interaction properties provided by a middleware architecture (e.g. transactional RPC communication). The author demonstrates the use of the specification model in more detail in the example introduced in the next section.

II.2.5 ADL tool support

Figure II.1, gives an overview of the ADL tool support needed so far. More specifically, the ADL is supported by an ADL compiler which accepts as input the textual description of an architecture and produces a symbol table that contains the same information in a form that can be systematically traced and manipulated by the rest of the ADL tools that are described in the following chapters. Moreover, the ADL compiler verifies that the axioms described in the `Base-Architectural-Style` theory hold for the given textual description. In addition to the ADL compiler, the ADL tool support comprises a parser for temporal logic specifications given in a STeP format. This parser is part of the STeP theorem prover.

II.3 A Case Study

This section presents a case study application, which is used throughout this document to exemplify the various steps of the systematic customization of middleware. The example is a distributed file system, consisting of individual file servers and individual clients accessing those servers. Typically, a file server manages a number of files and provides operations to clients for opening, closing, and updating files. Operations performed by a file server usually take place atomically and in isolation regarding multiple concurrently executing clients. However, a problem that regularly appears in such kind of systems is to be able to perform compound operations (i.e. sets of operations) on files that reside in different file servers atomically and in isolation regarding other concurrently executing clients. Typical examples are reliable workflow systems [90] in which clients are trying to execute a workflow over resources that are distributed over different file servers.

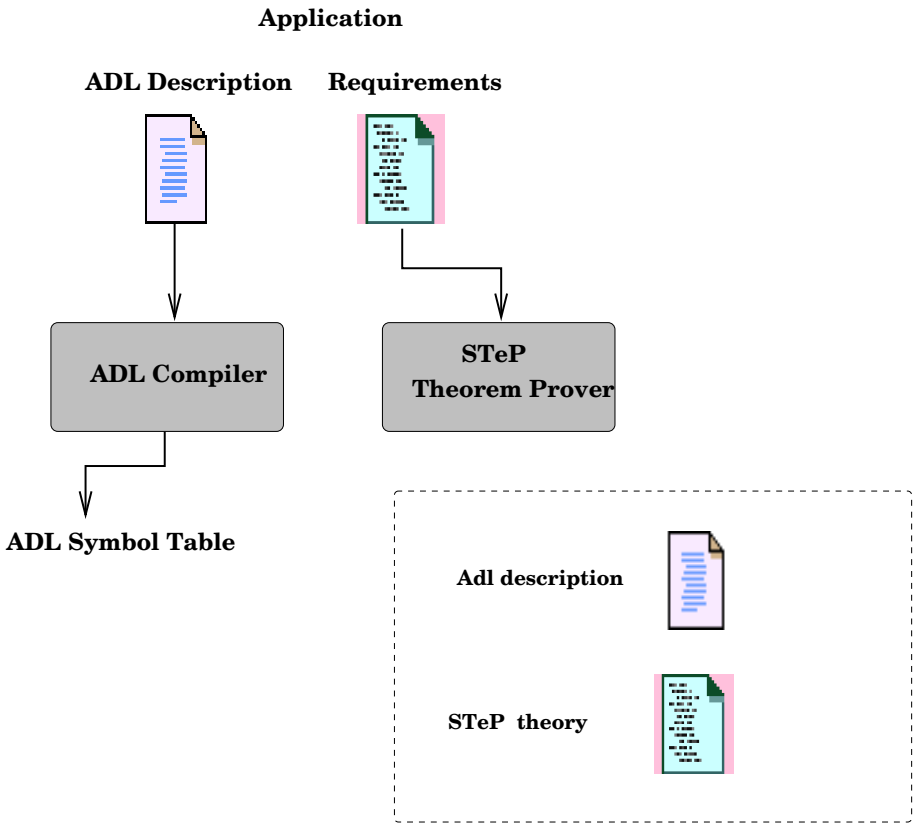

Figure II.1 ADL tool support : ADL Compiler

Hence, the distributed file system needs a middleware that mediates the interaction among clients and file servers, and further guarantees that the interaction is reliable, and provides means of performing compound operations atomically and in isolation. The remainder of this section addresses the difficulties of building such a middleware and justifies the need for a process for the systematic customization of middleware. Moreover, it exemplifies the use of the ADL, proposed in Section II.2, for describing the structure and the requirements of the case study application.

II.3.1 Motivating the case study

The ability to perform compound operations atomically and in isolation can be achieved using *transactions*. The concept of transactions has been around for quite a long time and is still considered as a very useful utility for building reliable systems. Based on the model proposed by Gray [24], a variety of others were developed in order to support application families (e.g. distributed databases, telecommunications, CAD/CAM) for which the traditional transaction model was too strict and costly. Well known examples of variations that originate from the traditional flat transaction model are those of nested [65], join-split [72],

and cooperative [66] transactions.

Due to the fact that transactions became a basic building block for a variety of application families, todays middleware infrastructures provide services that can provide transactions when combined into a middleware. Taking for example infrastructures like DCOM, EJB mentioned in Chapter I, we can derive that support for transactions is considered indispensable. Moreover, during the last years, there have been efforts to standardize the specification of services that can provide transactions when combined into a middleware. Examples of such standard specifications are the CORBA Object Transaction Service (OTS) [68] and the specification of the *transactions function* that is included in the RM-ODP standard.

Hence, when building a transactional middleware for the distributed file system, the architect is supposed to refine the abstract transactional requirements of the application into a concrete middleware architecture that satisfies them. Consequently, the architect must go through the different transaction models, pick up one that suits the application requirements and further refine the outcome in a concrete middleware architecture that can be implemented.

To simplify the work of the architect, the systematic customization of middleware proposes a systematic method for retrieving a concrete middleware architecture that refines the abstract requirements of an application. The above process is based on a structured repository of available middleware architectures, which is further detailed in the next chapter. The repository structure reflects the possible refinement relations among the available architectures.

The designer's work includes searching for middleware services, providing properties that match the ones described by the components that constitute the concrete middleware architecture. At this point, the issue is how much complicated would that be for the designer, especially given the additional help provided by the existence of standard service specifications. According to its specification [68], the OTS service may provide both flat and nested transactions, when combined into a middleware. Hence, the designer can start-up with studying the standard specification of a service and then go-on with searching for an available implementation of such a service. One usual trouble at this point is that specifications are in many cases rather imprecise and vague so as to satisfy different software vendors promoting already existing products. Moreover, when getting to the issue of available service implementations, it is a common case that services either partially comply to the standard, or fully comply to the standard and further provide extra features that could also be useful. Five available OTS implementations are examples that justify the above remark. The OTS implementation coming from BULL[3] does not support nested transactions; The OTS implementation coming from TRANSARC[4] does provide nested transactions but requires further support from software supported by TRANSARC; The OTS implemen-

[3]http://www-frec.bull.com:80/dom/orbtp/orbtp.html
[4]http://www.transarc.com/Product/Txseries/Encina/OTS/beta-notes.html

tation coming from CNET[5] provides flat, nested and open-nested transactions; The OTS service from ARJUNA[6] provides flat, nested transactions and additionally provides functionalities that enable to easily manage resources in a way transparent to the application. So, considering the case of an application that requires nested transactions, after retrieving an available implementation of OTS, based on the CORBA standard service specification, the designer has to verify that the service implementation does provide the properties it is supposed to provide. Considering the case of an application that requires flat transactions, after retrieving a set of available implementations, the designer has to select the one that better suits the application regarding other requirements (e.g. performance, scalability etc.). Performance, for instance, is typically among the basic requirements. Consequently, the designer is supposed to pick up a light-weight implementation of the OTS service and not one that gives extra features like open-nested transactions.

To help the work of the designer, the systematic middleware customization process offers a systematic method for retrieving middleware components that can be used to build a concrete middleware architecture. The proposed method is based on the middleware repository, which in addition to available middleware architectures, contains middleware components implementing services provided by a middleware infrastructure.

Summarizing, the designer has to study the properties of the available middleware implementations and evaluate those implementations in terms of other properties like performance, scalability etc. The latter task also aggravates the work of the developers, who are the ones that have to build different prototype middleware implementations that combine available OTS implementations with an ORB. Moreover, developers have to integrate the resulted prototype middleware implementations with the distributed file system application.

To help the developers, the systematic customization of middleware further proposes an automated method for integrating a retrieved middleware architecture within the architecture of a given application. The proposed method saves time from the developers, enables fast prototyping, and makes the evaluation of different implementations of a concrete middleware architecture a less laborious task.

II.3.2 Describing the application architecture

Figure II.2, gives the architectural description of the distributed file system application. The description includes two primitive component types, `FileClient`, `FileServer`, and a connector, named, `ReliableConnector`.

The `FileServer`, provides an interface of type `FileAccess`, that exports a set of operations for accessing files. The `FileClient` requires an interface of type `FileAccess`. The application requires the `ReliableConnector` to provide properties `Cmodel` for reliable RPC

[5]http://www.cnet.fr/a/cnet/welcome.html
[6]http://arjuna.ncl.ac.uk/OTSArjuna/index.html

```
interface FileAccess {
  void fopen (in string name, out handle file);
  void fread (in handle file, in long size, out sequence<octet> data);
  void fwrite (in handle file, in sequence<octet> data);
  void fclose (in handle file);
};

component FileClient {
  requires FileAccess FA;
};
component FileServer {
  provides FileAccess FA;
};
connector ReliableConnector {
  properties Cmodel, Tmodel;
};

configuration DFS {
 {
   instances
     FileServer Server;
     FileClient Client;
     ReliableConnector Connector;
   bindings
     Client.FA to Server.FA through Connector;
};
```

Figure II.2 Case Study: The ADL description of a distributed file system application

communication, and `Tmodel` for atomicity and isolation, as defined by the traditional flat transaction model. The architectural specification further defines a configuration, named `DFS`, which describes the overall structure of the distributed file system application. More specifically, the configuration defines two component instances, `Client` and `Server`, of the `FileClient` and `FileServer` types respectively. Moreover, it defines a connector instance, named `Connector`, of type `ReliableConnector`. The interface of type `FileAccess` required by the `Client` is bound to the corresponding interface that is provided by the `Server`, through the reliable connector. It is immediately visible that the architecture described so far is of a system that does not scale ; it consists of a single client accessing a single server. Typically, however, distributed file systems make use of a trader that serves to locate file servers based on requirements imposed by individual clients.

```
interface FileAccess {
};
interface QueryTrader {
  void query (in string name, in int load, out FileAccess FA);
};
interface RegisterTrader {
  void register (in FileAccess FA);
};

component FileClient {
  requires QueryTrader FT, FileAccess FA;
};
component FileTrader {
  provides QueryTrader FT, RegisterTrader RT;
  properties DynamicBinding;
};
component FileServer {
  provides FileAccess FA;
  requires RegisterTrader RT;
};
connector ReliableConnector {
  properties Cmodel, Tmodel;
};

configuration DFS {
  instances
     FileServer Server(j = 0, ...);
     FileClient Client(i = 0, ...);
     FileTrader Trader;
     ReliableConnector Connector;
  bindings
     Client(i).FT to Trader.FT through Connector;
     Server(j).RT to Trader.RT through Connector;
};
```

Figure II.3 Case Study: The ADL description of a scalable distributed file system application

Figure II.3, gives the architectural description of such a distributed file system application that scales in size. The ADL description includes a new component type named `FileTrader` providing two interfaces of types `RegisterTrader`, `QueryTrader`, which export operations for registering a file server in the trader, and querying the trader for a file server that possesses a specific file, respectively. The `FileTrader` component provides a property called `DynamicBinding`. This property states that if a file server calls the `register` operation provided by an interface of type `RegisterTrader`, the file server is registered to the trader. Moreover, if a client calls the `query` operation provided by an

interface of type `QueryTrader`, and there exists a registered file server that satisfies some required conditions regarding the load, the client is eventually bound to that file server. The `FileServer` component type, requires an interface of type `RegisterTrader` and provides an interface of type `FileAccess`. The configuration, named `DFS`, describes the overall structure of the application that contains a component instance of type `FileTrader`, a connector `ReliableConnector`, and an arbitrary number of instances of types `FileClient` and `FileServer`.

II.3.3 Describing the requirements of the application

The first of the application requirements is the one for reliable RPC communication. The temporal logic description of this property is given in the following STeP specification. More specifically, the following specification defines a property, called `Cmodel`, which roughly states that, if a request, req, is sent by a component, C, it is eventually delivered to its target, C', which may eventually return a response to C. Note that the definition of the `Cmodel` property uses the basic definitions given in the `Base-Architectural-Style` theory.

```
SPEC Cmodel-Dfs

include Base-Architectural-Style

value Conf : CONF


PROPERTY Cmodel
      ∀req : R,
      C, C' : C,
      rid : UID |
          Call(Conf, C, C', req, rid) ⇒
             ◇(UpCall(Conf, C, C', req, rid)∧
             ⊕□¬UpCall(Conf, C, C', req, rid)∧
             ⊖◻¬UpCall(Conf, C, C', req, rid)∧
             (∃resp : R |
                ◇(ReturnUpCall(Conf, C, C', resp, rid) ⇒
                    ◇ReturnCall(Conf, C, C', resp, rid))))
END
```

The remaining properties that describe atomicity and isolation from the traditional flat transaction model are specified similarly. The first step is to define a transaction. A transaction is characterized by a component, called transaction originator, that initiates and terminates a transaction, by calling the *Begin*, and *Commit* or *Abort* operations respectively. Hence, the originator requires at least an interface that contains those operations

and at some time exports requests that belong to the range of those operations. These requests are characterized by a unique identifier, $t.tid$, which is used to distinguish between consecutive transactions initiated by the same originator. Except for the definition of the originator and the requests to initiate and terminate a transaction, the definition of a transaction includes the set of requests issued in between calls to the *Begin*, and *Commit* or *Abort* operations.

The following theory defines formally the previously mentioned properties characterizing transactions. Note that the following theory is based on the definitions given in the `Base-Architectural-Style` theory. Moreover, it should be noticed that the following theory defines a new type called $\mathcal{T}$, which is used to describe properties of middleware architectures that provide transactions. Hence, the following theory *extends* the `Base-Architectural-Style` theory with features that allow to describe transactional interaction. In particular, the $TScope$ macro, states that the set of requests $t.RXID$ contains all requests issued by the originator between the calls for the initiation and termination of t. In addition, $t.RXID$ contains requests issued by other components while they were processing requests belonging to $t.RXID$.

```
THEORY Transactional-Style

type
```

$$\mathcal{T} = \{$$
$$O : \mathcal{C},$$
$$tid : \mathcal{UID},$$
$$reqB, reqA, reqC : \mathcal{R}$$
$$RXID : \{\{req : \mathcal{R}, rid : \mathcal{UID}\}\} \neq \emptyset$$
$$\}$$

```
value
```

$$Begin, Commit, Abort : \mathcal{O}_r$$

```
value
```

$$Conf : \mathcal{CONF}$$
$$Mdw : \mathcal{C}$$

```
macro
```

$$TScope(Mdw : \mathcal{C}, Conf : \mathcal{CONF}, t : \mathcal{T}) =$$
$$\forall req : \mathcal{R}, rid : \mathcal{UID} \mid$$
$$\{req, rid\} \in t.RXID \Leftrightarrow$$
$$(\exists C : \mathcal{C} \mid$$
$$DirectExp(Mdw, Conf, t, req, rid, C) \vee$$
$$(\exists rid', rid'' : \mathcal{UID},$$

$req', req'' : \mathcal{R}$
$C' : \mathcal{C} \mid$
$TransExp(Mdw, Conf, t, req, rid, req', rid', req", rid", C, C'))$

macro
$DirectExp(Mdw : \mathcal{C}, Conf : \mathcal{CONF}, t : \mathcal{T},$
$req : \mathcal{R}, rid : \mathcal{UID}, C : \mathcal{C}) =$
$\Diamond\Diamond(Call(Conf, t.O, Mdw, t.reqB, t.tid) \wedge$
$\Diamond(Call(Conf, t.O, C, req, rid) \wedge$
$\Diamond(Call(Conf, t.O, Mdw, t.reqC, t.tid) \vee$
$Call(Conf, t.O, Mdw, t.recA, t.tid)))))$

macro
$TransExp(Mdw : \mathcal{C}, Conf : \mathcal{CONF}, t : \mathcal{T},$
$req : \mathcal{R}, rid : \mathcal{UID},$
$req' : \mathcal{R}, rid' : \mathcal{UID}, req'' : \mathcal{R}, rid'' : \mathcal{UID},$
$C : \mathcal{C}, C' : \mathcal{C}) =$
$\Diamond\Diamond(DirectExp(Mdw, Conf, t, req', rid', C') \wedge$
$\Diamond(UpCall(Conf, t.O, C', req'', rid'') \wedge$
$\Diamond(Call(Conf, C', C, req, rid) \wedge$
$\Diamond(\exists resp : \mathcal{R} \mid ReturnUpCall(Conf, t.O, C', resp, rid'))))$

AXIOM Originator
$\forall t : \mathcal{T} \mid$
$(t.O \in Conf) \wedge$
$(\exists i : \mathcal{I}_r \mid$
$(i \in t.O.I_r) \wedge$
$(Begin, Commit, Abort \in i) \wedge$
$(t.reqB \in rang(Begin)) \wedge$
$(t.reqC \in rang(Commit)) \wedge$
$(t.reqA \in rang(Abort)))$

AXIOM Transaction-Scope
$\forall t : \mathcal{T} \mid TScope(Mdw, Conf, t)$

% atomicity, isolation macros

macro
$begin(Conf : \mathcal{CONF}, C : \mathcal{C}, t : \mathcal{T}) =$

$$
\begin{array}{l}
\quad \exists req : \mathcal{R} \\
\quad rid : \mathcal{UID} \\
\quad C' : \mathcal{C} \mid \\
\quad \{req, rid\} \in t.RXID \wedge \\
\qquad (UpCall(Conf, C', C, req, rid) \wedge \\
\qquad (\not\exists \{req', rid'\} : \mathcal{R} * \mathcal{UID}, \\
\qquad C'' : \mathcal{C} \mid \\
\qquad\quad \{req', rid'\} \in t.RXID \wedge \\
\qquad\quad \ominus \diamondsuit UpCall(Conf, C'', C, req', rid')))
\end{array}
$$

`macro`

$$
\begin{array}{l}
end(Conf : \mathcal{CONF}, C : \mathcal{C}, t : \mathcal{T}) = \\
\quad \exists req : \mathcal{R} \\
\quad rid : \mathcal{UID} \\
\quad C' : \mathcal{C} \mid \\
\quad \{req, rid\} \in t.RXID \wedge \\
\quad (UpCall(Conf, C', C, req, rid) \wedge \\
\quad (\not\exists \{req', rid'\} : \mathcal{R} * \mathcal{UID} \\
\quad C'' : \mathcal{C} \mid \\
\qquad \{req', rid'\} \in t.RXID \wedge \\
\qquad \oplus \diamondsuit UpCall(Conf, C'', C, req', rid')))
\end{array}
$$

`macro`

$$
\begin{array}{l}
involved(Conf : \mathcal{CONF}, C : \mathcal{C}, t : \mathcal{T}) = \\
\quad (t.O \in Conf) \wedge \\
\quad \exists C' : \mathcal{C}, \\
\quad req : \mathcal{R}, \\
\quad rid : \mathcal{UID} \mid \\
\quad \{req, rid\} \in t.RXID \wedge \\
\quad \diamondsuit \diamondsuit UpCall(Conf, C', C, req, rid)
\end{array}
$$

`macro`

$$
\begin{array}{l}
isolation(Conf : \mathcal{CONF}, t : \mathcal{T}) = \\
\quad \forall req : \mathcal{R}, rid : \mathcal{UID}, C, C' : \mathcal{C} \mid \\
\quad\quad involved(Conf, C', t) \wedge \\
\quad\quad \{req, rid\} \notin t.RXID \wedge \\
\quad\quad ((UpCall(Conf, C, C', req, rid) \wedge \oplus \diamondsuit begin(Conf, C', t)) \vee \\
\quad\quad\quad (end(Conf, C', t) \wedge \oplus \diamondsuit UpCall(Conf, C, C', req, rid)))
\end{array}
$$

```
macro
```
$$
\begin{array}{l}
\quad atomicity(Conf : \mathcal{CONF}, t : \mathcal{T}) = \\
\quad\quad \exists req : \mathcal{R}, \\
\quad\quad rid : \mathcal{UID} \mid \\
\quad\quad\quad (\{req, rid\} \in t.RXID) \wedge (failure(req)) \Rightarrow \\
\quad\quad\quad\quad \forall C : \mathcal{C} \mid \\
\quad\quad\quad\quad\quad involved(Conf, C, t) \wedge \\
\quad\quad\quad\quad\quad \exists \sigma : \Sigma \mid \\
\quad\quad\quad\quad\quad\quad \diamondsuit([C, \sigma] \wedge begin(Conf, C, t)) \wedge \\
\quad\quad\quad\quad\quad\quad \diamondsuit([C, \sigma] \wedge end(Conf, C, t))
\end{array}
$$
```
END
```

To specify the *atomicity* and the *isolation* transaction properties, two macros *begin* and *end*, are defined in the `Transactional-Style` theory. According to their definition, *begin* and *end*, hold respectively, at the time when a component C of $Conf$ joins the execution of a transaction t, and at the time when a component C stops participating in the execution of a transaction t. More specifically, the *begin* macro holds for a component C at the time when C imports, for the first time, a request belonging to $t.RXID$. The *end* macro holds at the time when C imports, for the last time, a request belonging to $t.RXID$. The *involved* macro is also used to define the *atomicity* and *isolation* macros. It holds for a component C if C is involved in the execution of t that takes place within a configuration $Conf$. Given the definitions of those macros the properties that describe the atomic and isolated execution of $t : \mathcal{T}$, within a configuration $Conf$ can be easily defined. According to the *atomicity* macro, if there exists a request belonging to $t.RXID$ indicating a failure, then all components involved in the execution of t, at the end of t, get into a state that is the same as the state they were into when they joined the execution of t. Similarly, according to the *isolation* macro, requests that do not belong to $t.RXID$ and are targeted to a component C' that is involved in the execution of t, are imported by C' either before the beginning or after the end of t.

Based on the `Transactional-Style` it is possible to describe the `Tmodel` requirements of the distributed file system application as given in the following STeP specification.

```
SPEC Tmodel-Dfs

include Transactional-Style

PROPERTY Tmodel
```
$$
\begin{array}{l}
\quad \forall t : \mathcal{T} \mid \\
\quad\quad (atomicity(Conf, t) \wedge isolation(Conf, t))
\end{array}
$$

```
END
```

Then in the next chapter those requirements are used to retrieve a middleware architecture that satisfies them.

II.4 ADL Evaluation

The ADL proposed for the systematic customization of middleware is evaluated based on the evaluation framework proposed by Medvidovic in [54] and presented in Section II.1.4. Different aspects of the ADL are evaluated throughout the chapters of this document, depending on their relevance to the issues of expressiveness, scalability, compositionality, refinement, and dynamic evolution. This section, discusses the issues of expressiveness, scalability, and compositionality.

As mentioned, an ADL should provide means for describing software in a simple, expressive, and understandable way. In the particular case of middleware, infrastructure-specific IDLs are used to partially describe the architecture of software built on top of the infrastructure. People that design and develop middleware are used to this kind of IDL descriptions, consequently, an ADL for the systematic customization of middleware should not be syntactically far from them. Among the ADLs examined into this document, ASTER provides a simple and understandable syntax that extends the basic CORBA IDL syntax with means that enable describing the structure of software. The ADL proposed in this thesis is based on the one previously proposed by ASTER. Hence, it is capable of describing software built on top of well-known existing middleware infrastructures, in a simple and understandable way. In should be further noticed that among our future intensions is to move from the ADL syntax that is currently used to a UML-OMG-like syntax [67], which will further simplify the description of software architectures. Regarding the rest of the ADLs examined in this document, we found the ones proposed by DARWIN and C2 the most user friendly, since they allow for both graphical and textual representations.

Architecture description languages should allow describing, developing and analyzing large-scale, real-world software systems. As mentioned in [54], ADL support for scalable systems can not be easily evaluated, the best proof is by case studies that exemplify the use of the ADL. From that prespective, most of the existing ADLs provide means that allow describing large-scale, real-world applications. For example, Wright was used in [3] to describe formally the HLA standard for component integration. Similarly, RAPIDE was used in [42] to describe the X/Open standard for distributed transactional processing. DCL and DARWIN were used to model CORBA. The ASTER ADL was used in [75] to describe basic features of software systems that conform to the standard for TINA applications [88]. The ADL proposed in this thesis is based on the basic concepts previously introduced by the ASTER framework and consequently is capable for describing large-scale real-world applications. A sample of the capabilities provided by the proposed ADL was given in Section II.3. In this example, the proposed ADL was used to describe two versions of a

distributed file system application. In the first version the configuration is fixed while in the second version the configuration potentially scales in size. The example is simplified (e.g. it could include a description of the Trading Service described in the CORBA standard, in place of the simple file trader that was given) and possibly misses several details. However, it is another hint, if not an evidence that the proposed ADL is capable for describing large-scale systems.

As expected, most existing ADLs are capable for describing the structure of large-scale real-world applications. What gets to be more difficult and imposes more problems is to model, analyze and verify the properties of large-scale, real-world applications. Automated formal analyses and verification tools suffer from problems like state explosion when they get to handle the description of large-scale systems [51]. In the case of the systematic customization of middleware, formal verification and exhaustive analysis of system properties is not among the objectives. Formal analysis and proof are only used towards verifying correctness of subsequent refinement steps, which is relatively feasible and does not suffer from typical problems caused by the complexity and the size of the system.

Finally, another important issue related to the scalability property is whether the proposed ADL allows for developing large-scale, real world applications. Most of the existing ADLs, targeted for software development come along with a software infrastructure and tools that allow to implement an architecture on top of this infrastructure. In the best case the underlying infrastructure is compliant to a well-known standard for middleware capable of supporting large-scale real-world applications. ASTER and DARWIN for instance come along with tools that allow to implement an architecture on top of a CORBA compliant middleware infrastructure. On the other hand, we have infrastructures like the ones provided by C2, DURRA etc. for which there is no proof that they do not suffer from any scalability problems. Besides from the previous remarks it worths noticing that all of the ADLs examined in this document, including the previous version of ASTER, provide support for developing a software architecture that is specific to the underlying middleware infrastructure. It is the author's opinion that an ADL should not be tight to any middleware infrastructure. Instead, a successful ADL should provide tools that allow to easily exploit and employ the features of different available middleware infrastructures. An ADL should provide support that allows to easily compare different implementations of an application, built on top of different middleware infrastructures. The aforementioned requirements are met by the ADL support proposed in this document, which aims at helping the development of applications on top of any typical middleware infrastructure dealing with interoperability, heterogeneinity, etc.

Regarding compositionality, and in the same spirit with the rest of the ADLs examined in this thesis, the proposed ADL allows to describe both primitive and composite components. Similarly, C2, CONIC, and DARWIN allow describing the architecture of composite components. In DCL, modules and protocols are groups of actors, and an actor can be a module, or a protocol. DURRA provides means to describe compound tasks, RAPIDE and SADL allow to map an architecture to another one. WRIGHT is the only ADL, which does

not provide specific constructs that enable compositionality. Conceptually, however, the hierarchical composition of architectures is accepted.

III Middleware Synthesis

The systematic middleware synthesis process comprises: (1) designing a concrete middleware architecture that refines some abstract requirements of a given application, (2) building an implementation of the resulting concrete middleware architecture that is made out of reusable middleware services. To accomplish the first step, a repository of available middleware architectures is proposed. The repository organization reflects the refinement relation among available middleware architectures, allowing to gradually refine abstract requirements into concrete middleware architectures. To accomplish the second step, this thesis proposes a systematic method for integrating the implementation of a concrete middleware architecture, out of reusable middleware services.

This chapter details the basic concepts towards performing the aforementioned steps. In order to give a clear view of what is the ultimate goal of the middleware synthesis, this chapter first highlights what is considered as a middleware architecture. Then, the refinement relation is precisely defined, based on previous work done in the field of software architecture refinement. Following, the structure and contents of the repository are detailed, together with the basic steps that constitute the systematic retrieval of a middleware architecture that refines some abstract application requirements. Finally, this chapter details the proposed method for assembling and integrating a middleware implementation within an application, given the architectural descriptions of the application, and of a concrete middleware architecture that meets the application requirements.

III.1 Middleware Architecture

Our goal is to design and implement a concrete middleware architecture that enables application components to interact in a way that satisfies their requirements. A brief description of the basic features characterizing a middleware architecture was already given in Section II.2.2. However, before giving details regarding the proposed middleware synthesis process, it is necessary give a more clear view on what is a concrete middleware architecture and what does it mean to refine abstract requirements of an application into a concrete middleware architecture.

III.1.1 Describing a concrete middleware architecture

According to the software architecture paradigm presented in Chapter II, the basic architectural elements that constitute a software architecture are components, and connectors that mediate the interaction among components. Components and connectors can be either composite, or primitive depending on whether, or not they are made out of other interconnected architectural elements. Following the very same paradigm, a middleware architecture is a sub-case of a software architecture and consists of components that interact through a connector. Consequently, the ADL, for the systematic customization of middleware, proposed in Section II.2, and used for describing the requirements and the structure of an application, is also used to precisely describe the properties and the structure of a middleware architecture.

As mentioned in Section II.2.2, a concrete middleware architecture comprises components, whose implementations correspond to client-side and server-side proxies. Client-side and server-side proxies combine functionality provided by a broker so as to enable application components to interact in a way that satisfies their requirements. Bindings between client-side and server-side proxies are realized through a more primitive connector like a primitive TCP/IP connector, or a primitive RPC connector. Two bound application components, one requiring an interface (i.e. a client instance) and one providing an interface (i.e a server instance) interact through a concrete middleware architecture as follows. The client issues a request to the client-side proxy of the server; the client-side proxy of the server forwards the request to the server-side proxy of the server; the server-side proxy of the server delivers the request to the server. Similarly, a response is returned back to the client. In order to realize the aforementioned scenario the following interfaces are provided, and required by a client-side proxy.

- For every interface provided by an application component, an interface of the same type is provided by the corresponding client-side proxy component. This interface is the one called by components of the application that require an interface of this type.
- a client-side proxy requires a broker-specific interface that allows it to forward requests made by client components to the server-side proxy.

Similarly, the following interfaces are required, and provided by a server-side proxy.

- For every interface provided by an application component, an interface of the same type is required by the corresponding server-side proxy component. This interface is used by the server-side proxy to forward requests to the application component.
- A server-side proxy provides a broker-specific interface used for delivering requests coming from the client-side proxy.

Except for the proxies, a concrete middleware architecture may comprise components that correspond to services provided by the underlying middleware infrastructure (e.g. security, transactions, etc). Middleware services can be used explicitly by the application, meaning that there exist client-side and server-side proxies, which allow application components to interact with the service. Similarly, middleware services can be used explicitly by other middleware services. In this case, one service plays the role of a client, requiring an interface that is provided by the other service, which plays the role of the server. Again, this interaction is realized through client-side and server-side proxies.

Functionality provided by a set of middleware services can be combined within a proxy together with functionalities provided by the broker so as to allow application components to interact in a way that satisfies the application requirements. For example, there exist middleware infrastructures such as ORBIX and MICO, which allow to execute operations within the client-side proxies, just before, or right after issuing a request. Moreover, they allow to execute operations within the server-side proxies, just before, or right after delivering a request to the server. If however, the previous flexibility is not provided by the underlying middleware infrastructure the remaining option for combining service functionalities with broker functionalities is with using components that wrap the implementation of a client-side proxy. This kind of components are widely known as *wrappers*. Using wrappers for combining functionality of a broker with functionality provided by other middleware services, so as to enable application components to interact in a way that satisfies their requirements, keeps the implementation of the client-side and server-side proxies clean. However, it burdens more the developers.

From the architectural point of view, a *wrapper* is very similar to a proxy. A wrapper consists of a client-side wrapper and a server-side wrapper. In principle, a client-side wrapper is used by a client component, to issue requests towards a server component. More specifically, the client-side wrapper forwards requests made by the client to the corresponding server-side wrapper. In addition, the client-side wrapper serves for calling operations provided by middleware services just before, or right after forwarding client requests to the server-side wrapper. The server-side wrapper delivers client requests to the client-side proxy of the server component. In addition, the server-side wrapper serves for calling operations provided by middleware services just before, or right after delivering a request to the client-side proxy of a server component.

In order to realize the previous interaction scenario, the following interfaces must be provided, by the client-side wrapper.

- For every interface provided by a client-side proxy, an interface of the same type is provided by the client-side wrapper.

Similarly the following interfaces must be required by a server-side wrapper.

- For every interface provided by a client-side proxy, an interface of the same type is required by the server-side wrapper.

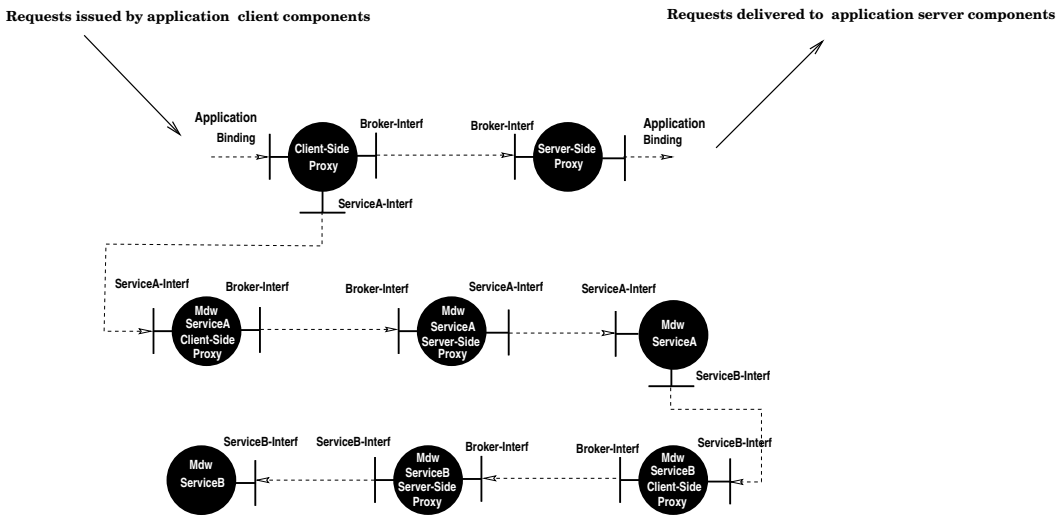

Figure III.1 A concrete middleware architecture

Figure III.1, gives an example of a concrete middleware architecture that mediates the interaction between application components that require an interface called `Binding` and application components that provide an interface `Binding` of the same type. This particular middleware architecture comprises a component, named `Client-Side-Proxy`, which implements a client-side proxy that can be called by client components. Moreover, the middleware architecture includes a component, named `Server-Side-Proxy`, which implements a server-side proxy that deliver requests to server components. The `Client-Side-Proxy` combines services provided by a broker and services provided by the `Mdw-ServiceA` component so as to be able to forward client requests to the server component. In addition the `Mdw-ServiceA` component explicitly uses the `Mdw-ServiceB` component.

III.1.2 Refining abstract requirements into a concrete middleware architecture

Refining an architecture into a more concrete one means to precise on the structure and properties of the elements that constitute the architecture. Hence, during an individual refinement step the architect may decompose an element of the abstract architecture into an architecture of more concrete architectural elements. To verify the correctness of this step, the architect must prove that the architecture of concrete elements provides the properties of the abstract element. The properties of the architecture is a combination of the properties provided by the individual concrete elements. This combination can be derived based on the way the elements are connected. According to Moriconi *et. al.*, [64], software architectures may be based on different architectural styles. Consequently, to refine a middleware architecture M' of a style S' into a more concrete middleware architecture M of a style S, the architect has to devise a a mapping I between the two architectures.

This mapping comprises:

- A *name mapping*, that associates instances of architectural elements that constitute M' with instances of architectural elements that constitute M.
- A *style mapping*, that maps basic style concepts, belonging to S', used to describe properties of M', to logical formulas, built out of basic style concepts belonging to the vocabulary of S.

In the context of this document, we assume that properties of a middleware architecture are described only using the base architectural style described in Section II.2.4, or extensions of this style. Hence, the style mapping is omitted from the middleware refinement process adopted here. Still according to Moriconi *et. al.*, and given two theories Θ and Θ', that describe properties of M and M', respectively, M refines M' if and only if the following hold:

$$\text{If } P \in \Theta' \text{ then } I(P) \in \Theta$$
$$\text{If } P \notin \Theta' \text{ then } I(P) \notin \Theta$$

Hence, in order to prove that M refines M', the architect has to verify that the interpretations of all the axioms defined in Θ' are logically implied, given the axioms defined in Θ. Verifying the second condition is, however, a much more complicated task. Roughly, the second condition states that M does not add any new properties about M'. Except for the difficulty to prove the second condition, it also seems that it is slightly too strict in the first place. Consider, for instance, the case of an RPC connector which further allows to execute reliable RPC calls atomically. This connector, according to the given definition of refinement, does not refine a connector that simply provides reliable RPC calls. Such a connector, however, could be used in the absence of one that only provides reliable communication. Based on those remarks, and given that software reuse is essential for reducing the cost of the software development process, we assume, from now on, that a middleware architecture M correctly refines another middleware architecture M', noted $M \underset{refines}{\Rightarrow} M'$, if and only if the following condition holds.

$$M \underset{refines}{\Rightarrow} M' \equiv \forall P' \in \Theta' \mid \exists P \in \Theta \mid P \Rightarrow I(P')$$

This condition for correct architecture refinement is the one traditionally used in ASTER. The basic inspiration comes from the work presented in [59], which was used to define the circumstances under which a concrete fraction of software refines another concrete fraction of software. The ultimate goal of the approach described in [59] was to structure a repository of available software in a way that enables the efficient retrieval of software. Again in this case, maximizing software reuse was among the primary concerns.

One last point that needs to be noticed arises when considering the gradual refinement process of abstract requirements of an application into a concrete middleware architecture that can be implemented. It is the author's opinion that the method, proposed by Moriconi *et. al.*, does not enable to clearly trace changes that take place during the very first step of the refinement process. More specifically, the authors assume that an abstract middleware architecture that is to be refined into a concrete one is complete with respect to its level of detail. Typically, however, application requirements come from customers and may be quite vague or imprecise. Moreover, customers usually seek for a perfect product and consequently their requirements reflect the ideal properties that the product is supposed to provide. Given the fact that in practice, and especially when considering software products, perfection is out of question, we end up that the requirements of an application may not be complete. For instance, there is no connector that guarantees absolute reliable message delivery. On the other hand, there may exist RPC connectors that do provide slightly weaker properties like at-most-once, or at-least-once delivery.

Hence, the very first refinement step, performed by the architect, comprises completing, or making more pragmatic, the requirements of the application. For instance, in the case where the application requirements describe some ideal middleware behavior, completion can be achieved based on the following pattern, which substitutes the ideal behavior with the disjunction of this behavior with the behavior that is expected in the case where either the software, or the hardware that the middleware relies on fails [94].

$$\texttt{value}\ \ failure :\rightarrow \{true, false\}$$

$$\begin{array}{l} I(ideal\text{-}behavior\text{-}specification)\ = \\ \quad (ideal\text{-}behavior\text{-}specification \vee \\ \quad\quad (failure \wedge non\text{-}ideal\text{-}behavior\text{-}specification)) \wedge \\ \quad (failure\text{-}cause\text{-}specification \Rightarrow failure) \end{array}$$

Given the remarks regarding middleware architecture and middleware architecture refinement it is now possible to proceed with the basic steps that constitute the systematic synthesis of middleware.

III.2 Middleware Retrieval

Given the refinement relation defined above, and in order to simplify the work of both the architect and the designer while trying to refine abstract requirements of an application into a concrete middleware architecture that satisfies them, the systematic customization of middleware relies on an organized repository of middleware architectures.

The repository can be systematically traced by the architect for retrieving a concrete middleware architecture that refines the requirements of an application. Moreover, the

repository can be used by the designer for retrieving implementations of concrete middleware architectures, or implementations of architectural elements that can be used to assemble the implementation of a concrete middleware architecture. This section details the structure of the proposed middleware repository and then presents the individual steps that make up the systematic retrieval of middleware.

III.2.1 Structuring a middleware repository

To be helpful for both the architect and the designer, the middleware repository is organized in two parts:

- A *design repository*, that contains the history of all the refinement steps that were performed by the architect, while developing middleware for applications that were built at some time in the past.

  The design repository comes along with a systematic process that allows the architect to locate (based on some given abstract application requirements) paths, i.e. sequences of refinement steps, which lead to concrete middleware architectures that possibly refine the abstract requirements.

- An *implementation repository*, that contains available implementations of :

  - Middleware components, implementing services provided by an existing infrastructure.
  - Concrete middleware architectures, that mediate the interaction between application components with respect to certain properties.

  The implementation repository comes along with a systematic process that allows the designer to locate middleware implementations.

Design repository

Using file system terminology, the design repository is a hierarchy of directories. Every directory in the hierarchy contains information for an architecture that mediates the interaction between application components. More specifically, this information comprises two files. The first file stores a textual description of the middleware architecture, described using the ADL proposed in Section II.2. The second file contains a formal specification of the properties provided by the middleware architecture. This formal specification is described using the basic architectural style that was given in Section II.2.4, or extensions of this style. In practice, a formal specification of middleware properties is given as a STeP theory that includes definitions given in the `Base-Architectural-Style` theory, or extensions of this theory. Except for these two files, a directory may contain links to available

implementations of the corresponding middleware architecture. Those links lead to the implementation repository, which is described right after. Finally, every directory has links to sub-directories. Every sub-directory stores information for a middleware architecture that refines the one described by the parent directory. Hence, sub-directories reflect information regarding previous refinement steps performed by the architect. Consequently, the whole directory hierarchy reflects a history of refinement steps that were performed by the architect during the design of middleware for applications, developed at some time in the past. The root of the directory hierarchy contains information regarding a middleware architecture that provides un-reliable RPC communication. Figure III.2, gives an abstract view of the middleware repository. In particular, the left side of the figure depicts an abstract view of the design repository.

Implementation repository

The structure of the implementation directory is simple. As mentioned, the implementation repository contains implementations of available middleware architectures and implementations of available middleware components. Its purpose is to help the designer when trying to retrieve an implementation of a concrete middleware architecture handed to him by the architect, or when trying to retrieve implementations of middleware components that can be used to construct a concrete middleware architecture that is not yet implemented.

Hence, the designer searches the implementation repository for something really specific, like a CORBA-based architecture, or a CORBA service. For that reason, the implementation repository is divided into a number of sub-directories, each one of which stores software based on a specific middleware infrastructure. For example we can imagine that the implementation repository contains a CORBA directory, a DCOM directory and a EJB directory. The CORBA directory includes a number of sub-directories, each one of which is specific to a particular CORBA compliant infrastructure (e.g. MICO, ORBIX etc.).

Every infrastructure-specific directory contains the implementation of the broker, and tools that ease the development on top of the particular infrastructure (e.g. IDL compilers etc.). Moreover, every infrastructure-specific directory is divided into two sub-directories. One of them stores information for components that implement services provided by the infrastructure. The other contains implementations of middleware architectures built out of those services.

In particular, the services sub-directory has a number of sub-directories, each one of which is specific to a particular service. Each service-specific directory contains a file with the ADL description of the service, a file that stores the formal specification of the properties provided by the service, and finally the code that implements the service.

The architectures directory stores different implementations of available middleware architectures into different directories, each one of which, has also links to directories in the

design repository that store the architectural information related to each implementation. The right side of Figure III.2 gives an abstract view of the implementation repository.

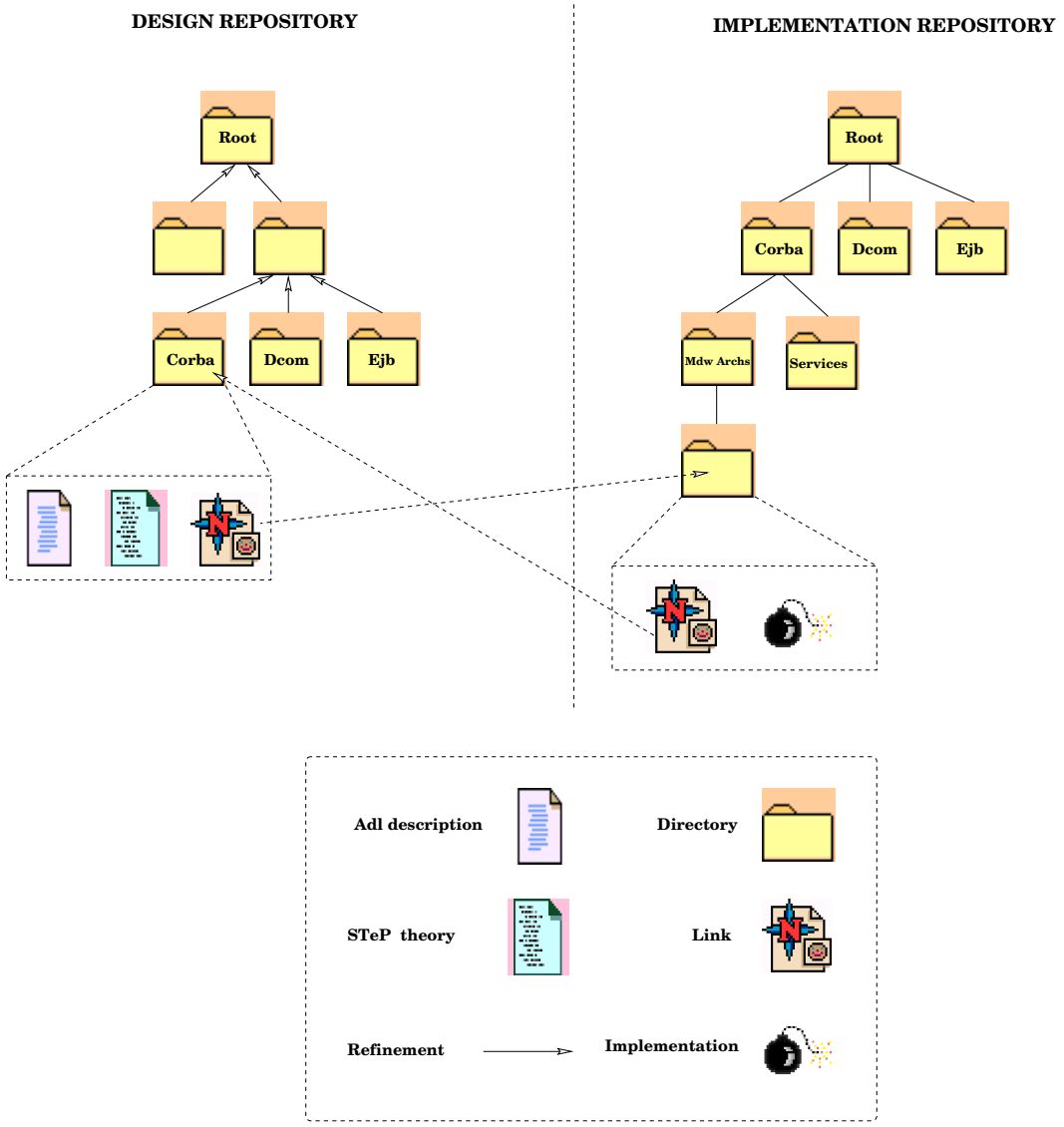

Figure III.2 An overall view of the middleware repository

III.2.2 Simplifying the work of the architect

Given the design repository, the architect is provided with an organized structure that contains all previous attempts to refine abstract requirements of applications into concrete middleware architectures. This knowledge can be a useful assistance towards future attempts for refining abstract application requirements. However, the size of the design

repository is expected to be large and hence, a simple and efficient method for navigating through the repository is needed. As mentioned, the goal of the architect is to refine abstract requirements into a concrete middleware architecture. Consequently, the architect should be provided with a method that allows him to easily locate refinement paths leading to concrete middleware architectures that satisfy the abstract requirements of an application. One simple approach for locating such paths is to start searching from the root directory for middleware architectures that refine the abstract requirements. However, in accordance with the definition of the refinement relation given in Section III.1.2, the previous simple search method involves proving that the properties of a middleware architecture imply the requirements of the application. Performing the previous task systematically requires using theorem proving technology. As it is widely known, theorem provers are not the easiest tools to cope with, and certainly they are not user friendly, neither efficient. In addition to those facts, the majority of theorem provers requires a lot of human intervention when proving complicated theorems. Thus, we arrive to a typical problem faced by most of the approaches for systematic software retrieval [59, 39, 76] that are based on formal specification matching [91]. Matching behavioral specifications with using a theorem prover, is usually equally hard to performing the matching manually. Trying to deal with the previous problem is hard since perfect theorem provers are not expected to exist in the near future. However, the typical approach is to try either to reduce, or completely avoid the use of a theorem prover during the retrieval of software. An example of an approach that falls into the latter case is presented in [21] where formal specification and theorem proving are used to construct a well-structured repository, which is then used for browsing. An approach that falls into the former case, is given in [77], where multiple filters are used during the retrieval of software. The first filter does signature matching, the second filter uses model checking to reject non-matching components as early as possible, and the last filter is a theorem prover that verifies correctness of the remaining results regarding the given requirements.

This last method seems to be interesting for the systematic customization of middleware. Typically, the first step towards automating the retrieval of software components is to define abstractions that characterize them. The two basic approaches for describing abstractions are *indexing vocabularies* and *formal methods* [22]. In other words, software components are characterized either by a formal specification that describes their behavior [91], or by values taken out of a controlled (i.e. there exist restrictions on how to describe a software component) [18] or an uncontrolled vocabulary [27, 28] (i.e. no restrictions on how to describe a software component). In the latter case, software retrieval is more efficient and less complicated. In the former case, retrieval gets more complicated but it is more accurate and precise. The approach proposed in [77] is based on a hybrid method (function signatures are out of an indexing vocabulary, while behavior is described using a formalism) for characterizing software components, which reduces the use of a theorem prover.

A method inspired from the one proposed in [77] is introduced for reducing the use of a theorem prover while trying to locate paths in the design repository that lead to

concrete middleware architectures, which refine the requirements of an application. More specifically, properties provided by a middleware architecture are described based on the basic architectural style described in Section II.2.4, or extensions of that style. Similarly, application requirements are described based on the basic architectural style, or extensions of that style. Then, a middleware architecture may refine application requirements if the description of its properties is based on the architectural style used to describe the application requirements, or an extension of that style. For example, a middleware architecture may refine requirements defined based on the `Transactional-Style` only if its properties are described based on the `Transactional-Style`, or an extension of the `Transactional-Style` style. Based on those remarks, locating abstract middleware architectures that are gradually refined into concrete middleware architectures, which satisfy requirements of an application comprises two steps:

(1) Locating abstract middleware architectures whose properties are described using, at least, the same style as the one used for describing application requirements.

(2) Then, starting from those middleware architectures, try to locate one that refines the application requirements based on the definition for correct architecture refinement given in Section III.1.2.

Performing the first step

From a practical point of view, properties of a middleware architecture are given as a number of axioms defined within a STeP theory. In order to describe those axioms, the architect uses, at least, the definitions of basic architectural elements and relevant concepts, which are given within the `Base-Architectural-Style` STeP theory. To be able to use those basic definitions the architect is obliged to include the `Base-Architectural-Style` STeP theory, within the theory of the middleware architecture using the STeP specific `include` statement. In the most general case the architect is obliged to include an extension of the `Base-Architectural-Style` STeP theory. This `include` statement serves as a search key when trying to accomplish the first step of the middleware retrieval.

More specifically, given a STeP theory Θ that corresponds to the architectural style used to describe application requirements, and starting from the root directory of the design repository, the search algorithm performs the following steps:

(1) If the STeP theory in the current directory includes Θ, or a theory that includes Θ, return the current directory as a result and terminate the recursive step.

(2) Else, for all sub-directories of the current directory, perform the previous step.

The outcome of the previous algorithm is, thus, a number of directories that contain information for middleware architectures, which may refine the abstract requirements of the application.

Performing the second step

To accomplish the second step of the middleware retrieval, the traditional ASTER approach is used [39]. More specifically, for each directory returned from the previous step, the following algorithm is followed:

(1) If the middleware architecture, described by the information stored in the current directory, refines the requirements of the application, return the current directory and terminate the recursive step.

(2) Else, perform the previous step, for all sub-directories of the current directory.

Hence, the result of this algorithm is a set directories describing middleware architectures that refine the application requirements. Each one of those directories is the root of a directory hierarchy, whose leaves may describe concrete middleware architectures that satisfy the application requirements. From this point after, it is up to the architect to pick up one, or more concrete architectures and hand them to the designer.

At this point it should be mentioned that if either of the previous steps fails to return a result, then there exists no middleware architecture within the design repository that may be refined into a concrete middleware architecture that satisfies the application requirements. Consequently, the architect has to design the required concrete middleware architecture. During the design process, the architect typically performs several subsequent refinement steps resulting in middleware architectures that must be stored within the repository in a way that preserves the refinement relation. Starting from the application requirements, the architect must insert in the design repository the description of an abstract middleware architecture that mediates the interaction between application components in a way that satisfies the application requirements. Possibly the previous description would include only the required properties, as given within the ADL description of the `connector` required by the application.

To successfully perform the above step, the architect must create a new directory that contains information about the new middleware architecture. Then, the architect must locate a number of directories within the design repository, describing middleware architectures for which the following conditions hold:

- The application requirements refine the properties of the middleware architecture.
- The middleware architecture is not refined into another one whose properties are refined by the application requirements.

Finally, the architect must create sub-directory links within the previous directories, leading to the new directory.

III.2.3 Simplifying the work of the designer

Following a typical software development process, the application designer gets as input from the architect, at least one directory that stores information regarding a concrete middleware architecture. The designer's main responsibility is to realize this architecture, possibly taking into account requirements regarding performance, scalability etc. The first thing checked by the designer is whether the directory, handed to him by the architect contains links leading to available implementations of the concrete middleware architecture, stored in the implementation repository. If this is the case, the designer further checks whether the available implementation meets his requirements on performance, scalability etc. Again if this is the case, the designer's work is finished. Then, it is the developer who takes over integrating the available implementation within the application. It must be noted at this point that the systematic customization of middleware does not provide, so far, any help to the designer, when trying to verify properties like scalability and performance, against application requirements. However, it is among the current objectives of the ASTER project to elaborate solutions to this issue. More specifically, [13] discusses extensions in the implementation repository that enable to structure available implementations of middleware architecture based on criteria like performance, algorithmic complexity, exchangeability, etc.

Getting back to the designer's responsibilities, if there exists no available implementation, or if the implementation does not meet the requirements on performance, scalability etc., the designer is obliged to design the implementation himself. This process involves designing the implementation of the individual architectural elements that make up the architecture. However, it is possible that the implementation repository contains implementations of some of those architectural elements. Hence, before starting to design the individual architectural elements from scratch the designer can look up in the repository for those elements. Since the designer is provided with the description of the concrete middleware architecture, he is aware of the properties that characterize the observable behavior of the individual elements that constitute it. Moreover, since the middleware architecture is a concrete one, the designer can reason about whether he is after architectural elements that implement CORBA, DCOM, or EJB services. Hence, searching in the implementation repository for available implementations that can be reused to construct the implementation of a concrete middleware architecture takes place according to the following steps:

(1) Go to the services directory of the middleware infrastructure that the concrete middleware architecture relies on.

(2) For every middleware component, representing a service, required for constructing the concrete middleware architecture do:

   (A) Locate middleware components whose properties are described using, at least, the same style as the one used for describing the properties of the required middleware component.

(B) Starting from middleware components resulting from the previous step, select the ones whose properties imply the ones of the required component. In other words, select middleware components that refine the one required for constructing the concrete middleware architecture.

Given the retrieved implementations of the elements that can be used to construct the implementation of a concrete middleware architecture, the developer takes over for actually building the implementation. Finally, it must be noted that once a new implementation of a concrete middleware architecture is built, the designer inserts it in the implementation repository.

III.2.4 ADL tool support

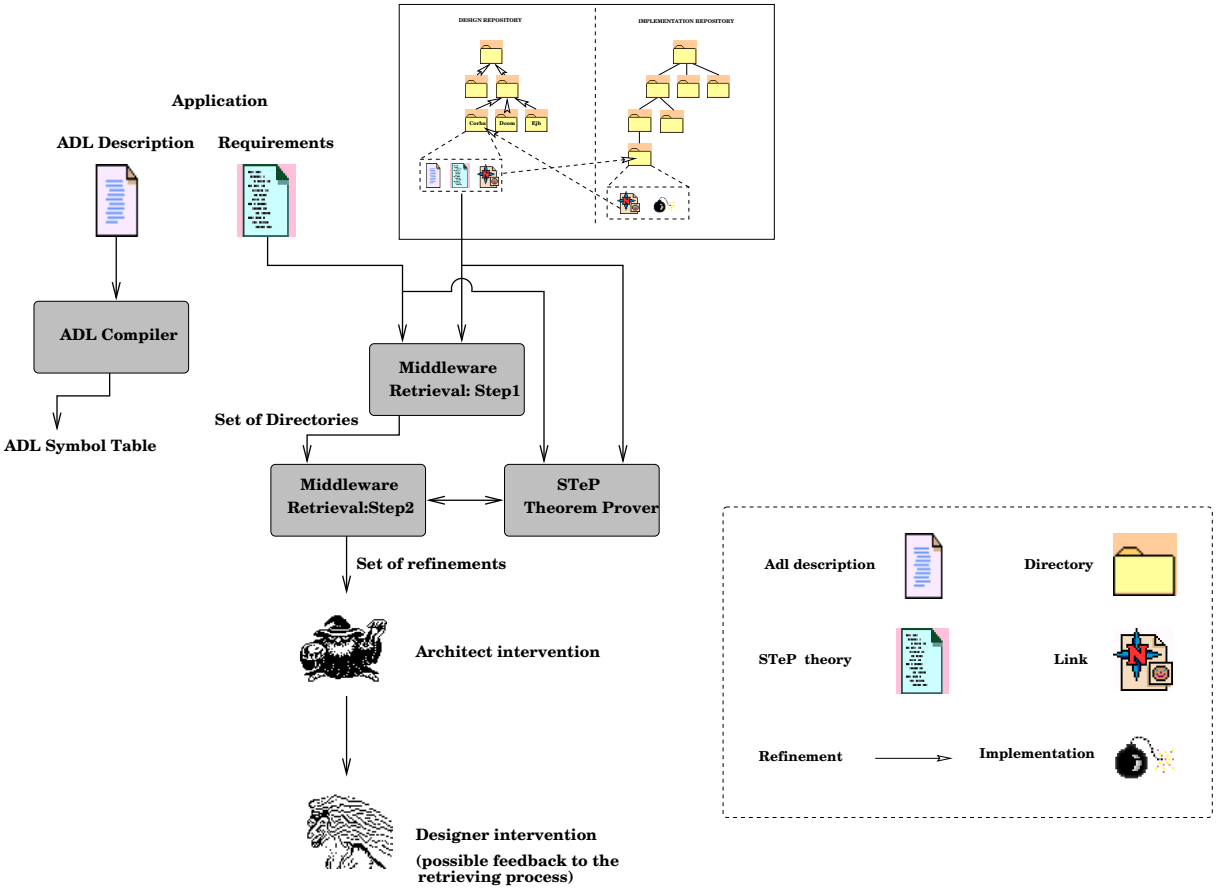

Figure III.3 ADL tool support for middleware retrieval

Summarizing this section, Figure III.3 gives an overall view of the ADL tool support, which now additionally comprises tool support for the systematic middleware retrieval.

More specifically, the middleware retrieval algorithm takes as input a STeP file that describes application requirements and uses the STeP files stored in the design repository to perform the first step of the algorithm given in Section III.2.2. Those files are input to the STeP theorem prover, which is used to perform the second step of the algorithm. Similarly, the previous algorithm and the STeP theorem prover are used to ease the designer's work when searching for available implementations of elements, which can be used to build a concrete middleware architecture.

III.3 Middleware Integration

The integration of a middleware implementation that realizes a concrete middleware architecture comprises assembling the implementations of the individual elements that make up the concrete middleware architecture, and combining the resulting implementation with the application. This work is typically performed by the application developers based on the input they get from designers. In the best case, where an implementation of a concrete middleware architecture was found in the implementation repository, the developer has to combine it with the application. In the worst case, where no implementation was found, the developer has to build the concrete middleware architecture using existing middleware services that implement components that constitute the architecture.

This section proposes a systematic method for assembling the implementation of a middleware architecture, that can be easily combined with any given application. This method simplifies the work of the developers, enables fast prototyping, and reduces the cost of evaluating different implementations of an application built on top of different middleware infrastructures. More specifically, this section discusses first the work of the developer when building a concrete middleware architecture. Part of the resulting implementation (e.g. proxies and wrappers for the application components) is specific to the application. However, building those parts for different applications involves repeating the very same process multiple times. Based on this remark, this section proposes using a simple language for describing the process of developing the application-specific parts of the middleware, in way independent from the application.

III.3.1 Simplifying the work of the developer

Let us now take a closer look at the developer's work. Taking the worst case mentioned above, suppose that the implementation repository does not contain the implementation of a concrete middleware architecture. Then, the input from the designer comprises a number of components implementing middleware services provided by a middleware infrastructure. Then, according to Section III.1, the developer has to perform the following tasks:

- Build client-side and server-side proxies for middleware services that are used within the concrete middleware architecture.

- Build wrappers that wrap client-side proxies and combine functionalities provided by middleware services, so as to enable application components to interact in a way that satisfies their requirements.
- Build client-side and server-side proxies that combine functionality of the broker and possibly functionality of the services, so that application components interact in a way that satisfies their requirements.

Let us now examine in turn each one of the aforementioned tasks.

Building client-side and server-side proxies for middleware services

Most of the existing middleware infrastructures provide tools, which, given a specific input from the developer, generate client-side and server-side proxy implementations automatically. For example, all CORBA compliant middleware infrastructures provide a CORBA IDL compiler, which apart from syntactically checking the IDL descriptions of the elements that make up an application, also generates the implementations of the corresponding proxies. Similarly, DCOM provides the MIDL compiler that generates the code of the client-side and server-side proxies, given the MIDL description of an application. JAVA RMI comes along with the RMIC compiler that generates client-side and server-side proxies for an application starting from the specification of the interfaces provided by the application.

Hence, the developer's work amounts to producing input for the aforementioned tools. In the absence of a tool that automatically generates client-side and server-side proxies the developer has to produce their implementation manually. Nevertheless, building client-side and server-side proxies for middleware services, used within a concrete middleware architecture, is done only once, when assembling for the first time the implementation of a concrete middleware architecture. Then, every-time this implementation is reused in the context of a different application, the developer's work is trivial.

Building wrappers

Briefly summarizing what was stated in Section III.1, a wrapper to a client-side proxy is divided in two parts, the client-side wrapper and the server-side wrapper. Then, for every interface provided by a client-side proxy, an interface of the same type is provided by the client-side wrapper. For every interface provided by a client-side proxy, an interface of the same type is required by the server-side wrapper. Given that the interfaces provided by the client-side proxy depend on the application, the same holds for those provided by the client-side wrapper and those required by the server-side wrapper. Moreover, according to the interaction scenario given in Section III.1, the implementation of a server-side wrapper calls the corresponding client-side proxy. Thus, the implementation of the server-side wrapper depends on the application.

Moreover, it should be noted that when building wrappers that enable application components to interact in a way that satisfies their requirements, the developer roughly follows the next steps. For every different client-side proxy, the developer implements a corresponding client-side wrapper, and a corresponding server-side wrapper. The implementations of the different client-side wrappers call the same operations just before and right after calling the corresponding server-side wrapper. The implementations of the different server-side wrappers call the same operations just before and right after calling the corresponding client-side proxy. Hence, the developer repeats the very same process when implementing each one of the wrappers except that each time, the wrapper's interface and the interface of the client-side proxy that the wrapper calls are different and both depend on the application.

Hence, when building the implementation of a concrete middleware architecture for the first time, the developer has to build the application-dependent parts. Moreover, whenever reusing the implementation of a concrete middleware architecture within the context of another application, the developer has to rebuild the application-dependent parts. Fortunately, the application-dependent parts of the middleware do not relate to the application implementation; instead they only relate to the architecture of the application (e.g. component types, interfaces, etc). Given the above remarks and in order to ease the work of the developers, this thesis proposes a systematic method for building application-dependent parts of a middleware implementation that can be easily reused.

The basic idea behind the proposed method is to provide the developer with a simple language, which enables him to describe abstractly, i.e. independently from the specific application, the process of building the application-dependent parts of the middleware. The description of this process is specified at the time when the implementation of a middleware architecture is built for the first time. This process description, together with a code generator that additionally takes as input information coming from the ADL description of a particular application, are used to generate the application-dependent parts of the middleware, every-time this middleware is reused for a different application. This idea was first proposed in [93] and was further elaborated in [92].

Building client-side and server-side proxies for application components

To combine a middleware implementation with an application, the developer has to either implement client-side and server-side proxies that enable interaction among application components, or to provide input to infrastructure-specific tools that generate the client and server side proxy implementations. The proxy interfaces and implementations are application-dependent. Consequently, the input to the tools that generate them automatically is application-dependent. Again, the application-dependent parts of the proxy implementations relate only to the architecture of the application (e.g. component types, interfaces etc.).

In addition to the above remarks, when implementing proxies, or when producing input

for tools that generate proxies for the different components that make up a particular application, the developer follows the very same process. For example, when producing input for a CORBA IDL compiler the developer performs the following steps. For every component type described in the ADL description of the application, a corresponding IDL description should be generated. More specifically, the resulting IDL description defines a CORBA `Module` for every different component type. This module should contain CORBA IDL interface descriptions for every interface provided by a particular component type, etc.

Based on the idea introduced above, it is possible to simplify the process of implementing, or generating proxy implementations. In the absence of a tool that automatically generates proxy implementations, the developer uses the proposed language to abstractly describe the process of implementing middleware proxies for application components. In the presence of tools that automatically generate proxy implementations, the proposed language can be used for describing the process of producing input to those tools. Then, every time the middleware architecture is reused for mediating the interaction among components of an application, the process description is fed to a code generator that generates the proxies, or input to the tools that generate proxies, based on information coming from the ADL description of the particular application.

III.3.2 Development process directives

Summarizing so far, this thesis proposes using a simple language to describe a *process* that must be followed to develop the application-specific parts of a middleware. Let us now examine in more detail the requirements that the proposed language must meet.

In general, the proposed language must enable the developer to describe a process that iterates through the basic architectural elements that constitute the application (e.g. components, configurations etc.), or a process that iterates through basic features (e.g. interfaces, operations etc.) that recursively characterize those basic architectural elements, and describe source code that must be generated for each one of those. Hence, the *object* of a process that describes the way to develop the application-specific parts of the middleware is either a basic architectural element, or features that recursively characterize a basic architectural element.

Moreover, the process language must provide means that enable the developer to describe a process that tests whether or not, certain process objects are present in the architectural description of the application, and depending on the case produce a fraction of source code. For example, the developer may want to describe a process, which tests whether or not, an operation has a return value. Depending on the case, the previous process either generates or not the declaration of a variable used to store the return value.

Every object within the ADL description of the application is named. More specifically, component types have names. The same holds for interface types, operations, arguments, configurations etc. Hence, the process language must provide means that allow describing a

```
Input             ::=  <TextStream>
TextStream        ::=  <TextStream> <PlainText>
                    |  <TextStream> <PlainPrefix>
                    |  '(' <TextStream> ')'
                    |  <TextStream> <Process>
PlainText         ::=  '$'
                    |  '.'
                    |  'Iff'
                    |  'Iterate'
                    |  'Not'
                    |  [ \t\n]*
                    |  Id
Process           ::=  '$' '(' <ProcessBody> ')'
ProcessBody       ::=  <ProcessObject>
                    |  <ProcessNegation>
                    |  <ProcessIteration>
                    |  <ProcessCondition>
ProcessObject     ::=  Id
                    |  <ProcessObject> '.' Id
ProcessNegation   ::=  'Not' <Process>
ProcessIteration  ::=  'Iterate' <Process> Id <TextStream>
ProcessNegation   ::=  'Iff' <Process> <TextStream>
```

Figure III.4 The grammar of the development process language

process that generates source code, which contains names characterizing a process object. For example the name of a particular component type described in the ADL description of the application is needed to generate a corresponding CORBA module. Similarly, the name of a particular interface type described in the ADL description of the application is needed to generate a corresponding CORBA IDL interface. For the purpose of this thesis, a minimal extensible process language was developed. The grammar of this language is given in Figure III.4.

The language grammar accepts as input a text stream that corresponds to a fraction of source code which occasionally contains development process directives. A directive starts always with the '$' character and it is enclosed within parentheses. The process directive can be of four different types: an object, an iteration, a condition or, a negation.

The *object directive* is given in the following syntax:

```
$(<ProcessObject>)
```

The object directive instructs the code generator to access the value of a process object. An object can be a list of application-specific basic architectural elements. described in the ADL description of an application. For instance, an object may be a list of basic component types described in the ADL description of an application. A process object may be a list of features (or a single feature) that recursively characterize a basic architectural element. For instance, an object may be a list of interfaces required by a component type, or the name of a component type. Finally, an object may be the name that identifies a particular feature described in the ADL description of the application.

In practice, a `<ProcessObject>` is reduced into a sequence of identifiers, which are out of a fixed set of keywords recognized by the code generator. Valid sequences of identifiers, i.e. sequences of identifiers that are meaningful for the code generator, identify a list of basic architecural elements, or a list of features, or a single feature that recursively characterize a basic architectural element. Figure III.5 shows valid sequences of keywords that can be used to describe a developement process for a component type. For example, `$(component)` instructs the code generator to access a list of component types described into the ADL description of an application. `$(component.requires)`, instructs the code generator to get the list of required interfaces. `$(component.requires.interface)`, instructs the code generator to access the type of a particular interface required by a component.

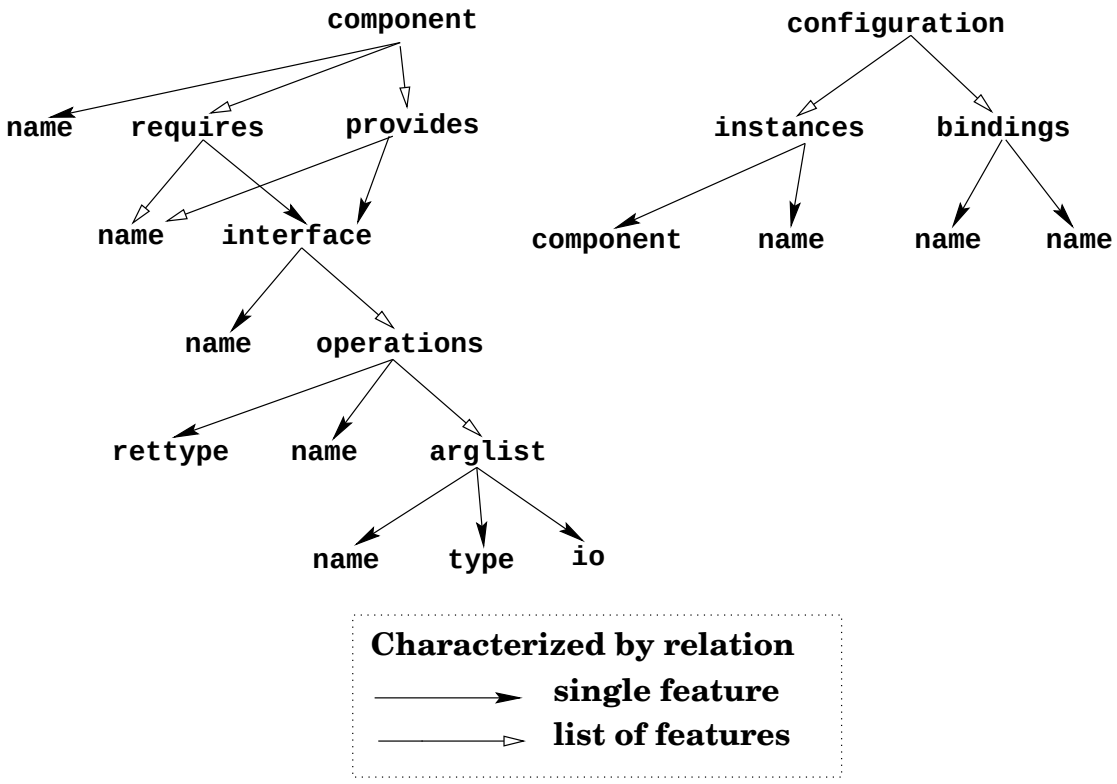

Figure III.5 Sequences of keywords identifying features that recursively characterize a component type

In the particular case where a process object is just a name, the code generator is instructed to just generate this name. For example, `$(component.name)` instructs the code generator to generate the name of a particular component type.

The *iteration directive* is given in the following syntax:

```
$(Iterate <ProcessObject> Id <SourceCode>)
```

This directive instructs the code generator to iterate through a list, `<ProcessObject>`, of objects and generate `<SourceCode>` for each one of those objects. Note that in the general case, `<ProcessObject>` can be a directive. This allows to define nested iteration directives.

For example, the following directive instructs the code generator to iterate through the different component types described in the ADL description of an application and generate a CORBA module named with the name of the corresponding ADL type.

```
// Corba module definition
$(Iterate $(component) component
module $(component.name) {
// module body
}
)
```

The *condition directive* is given in the following syntax:

```
$(Iff <ProcessObject> <SourceCode>)
```

This directive instructs the code generator to produce `<SourceCode>` if a list of objects `<ProcessObject>` is present in the ADL description of an application.

For example, the following directive instructs the code generator to generate a CORBA module, only for components that provide at least one ADL interface. Each module contains CORBA IDL interface definitions for all the interfaces provided by a particular ADL component type.

```
// Corba module definition
$(Iterate $(component) component
  $(Iff $(component.provides)
    module $(component.name) {
    // module body
    $(Iterate $(component.provides) provides
     // Corba interface definition
     interface $(provides.interface.name) {
     // interface body
     }
    }
    )   ) )
```

Note that in the general case, `<ProcessObject>` can be a directive. This allows to define nested condition directives, or combine the condition directive with other directives.

The *negation directive* is given in the following syntax:

```
$(Not <ProcessObject>)
```

The negation directive simply instructs the code generator to negate the absence, or the presence of `<ProcessObject>`. Note that again, in the general case `<ProcessObject>` can be a directive. This allows to define nested negation directives, or to negate the outcome of a condition directive.

More detailed examples describing the use of the developement process directives are provided in the case study section that follows.

III.3.3 ADL tool support

Summarizing this section, Figure III.6 gives a complete view of the ADL tool support for the systematic synthesis of middleware. In addition to the tools depicted in Figure III.3, the ADL tool support now comprises a code generator that takes as input application-specific ADL information, as extracted by the ADL compiler, mentioned in Section II.2.5, and the implementation of a concrete middleware architecture, parts of which are development process directives. Based on the ADL information, the code generator generates the implementation of the application-dependant parts of the concrete middleware architecture.

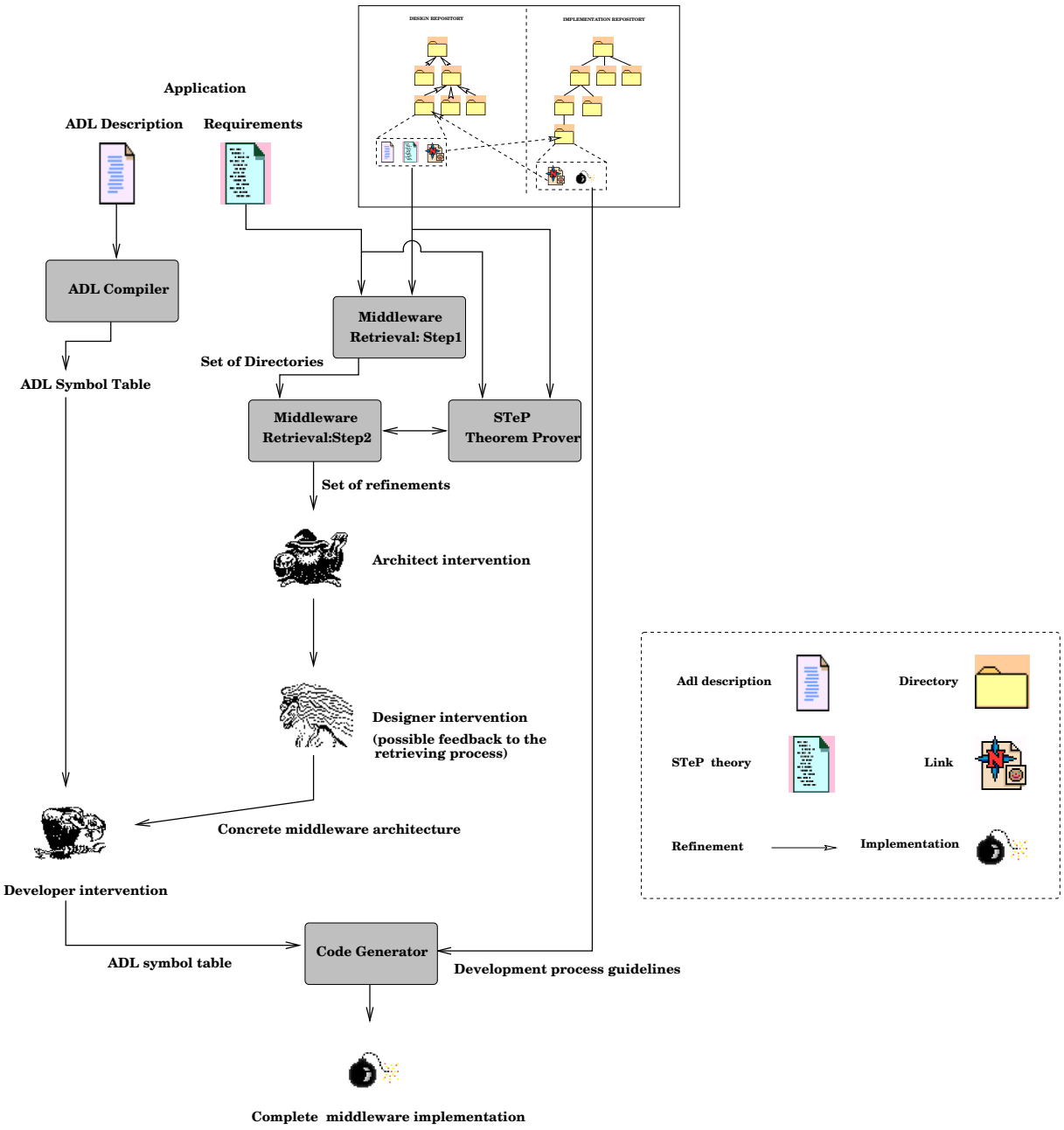

Figure III.6 ADL tool support for middleware integration

III.4 A Case Study

This section details the overall middleware synthesis process with an example, based on the scenario presented in Section II.3. The goal is to refine the requirements of the application into a concrete middleware architecture that satisfies them, and integrate the

resulting middleware imlpementation within the application. The requirements imposed by the distributed file system application comprise those for reliable RPC communication and transactions.

III.4.1 Middleware repository

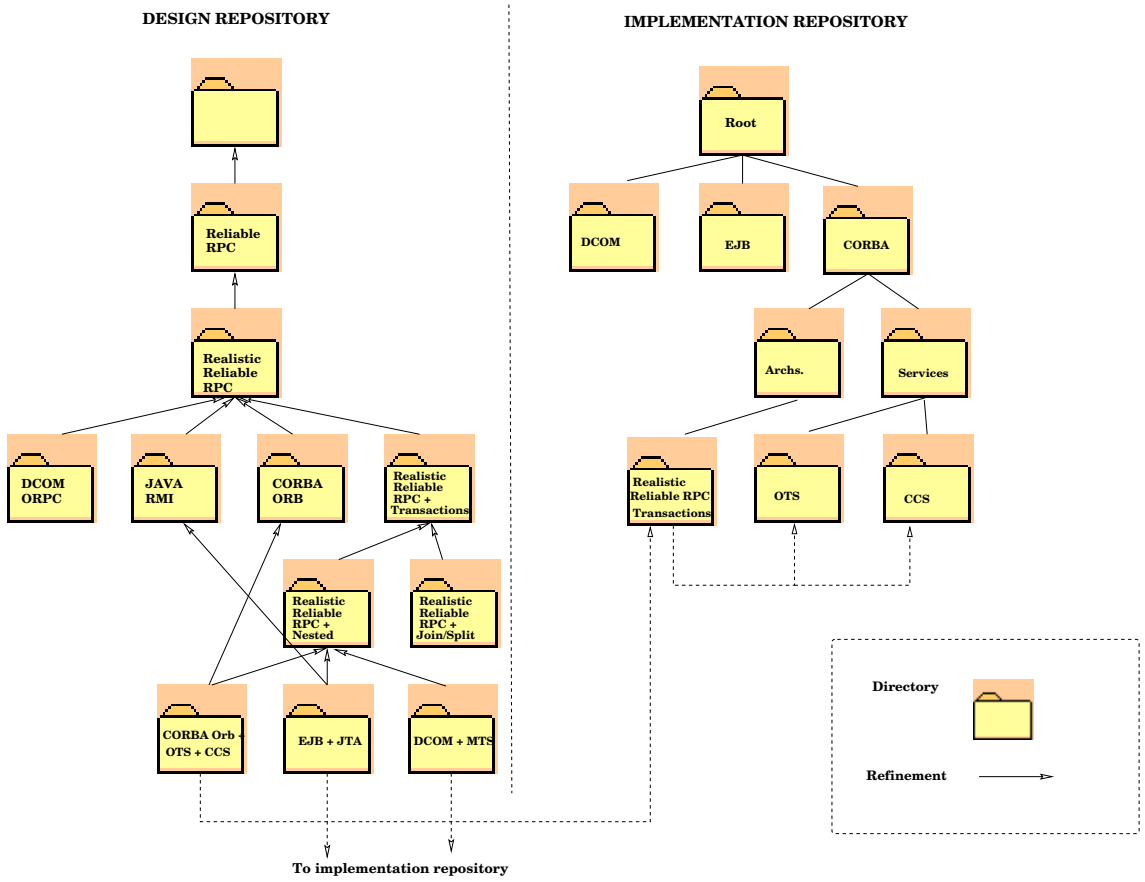

Figure III.7 Case Study: The middleware repository

Figure III.7, depicts the middleware repository used to exemplify the middleware synthesis process. As described in Section III.2.1 the repository is divided into a design repository and an implementation repository. The design repository stores a history of refinement steps performed at some time in the past. The implementation repository contains implementations of concrete middleware architectures and, implementations of available brokers and services. The root directory of the design repository contains the ADL description of a middleware architecture that provides un-reliable RPC communication. The root directory contains a number of sub-directories, which store information about middleware architectures that refine the one described in the root directory. More specifically, the root architecture is refined into a middleware architecture that provides reliable RPC

communication. The directory representing the architecture that provides reliable RPC communication includes the following theory that describes the properties guaranteed by the architecture. Note that the following theory is based on the definitions given in the `Base-Architectural-Style` theory.

```
THEORY Reliable-Rpc-Communication

include Base-Architectural-Style

value Conf : CONF

AXIOM Reliable-Rpc
      ∀req : R,
      C, C' : C,
      rid : UID |
          Call(Conf, C, C', req, rid) ⇒
             (◇(UpCall(Conf, C, C', req, rid)∧
             ⊕□¬UpCall(Conf, C, C', req, rid)∧
             ⊖□¬UpCall(Conf, C, C', req, rid)∧
             (∃resp : R |
                ◇(ReturnUpCall(Conf, C, C', resp, rid) ⇒
                   ◇ReturnCall(Conf, C, C', resp, rid)))))
END
```

Getting to the next level of refinements, the architecture that provides reliable RPC communication is refined into an architecture that provides realistic reliable RPC communication. This architecture consists of a client and a server side proxy connected through a connector that guarantees realistic reliable RPC communication. The properties guaranteed by this middleware architecture are given in the theory that follows. Note again that the theory that follows is based on definitions given in the `Base-Architectural-Style` theory. The properties described in `Realistic-Reliable-Rpc-Communication` theory are obtained with applying the completion refinement pattern detailed in Section III.1.2 on the properties provided by the previous middleware architecture. The `Realistic-Reliable-Rpc` axiom contains two clauses. The first clause describes request delivery in the ideal case where there is no failure on neither the software, nor the hardware that the middleware architecture relies on. This first clause is identical with the `Reliable-Rpc` axiom described in the `Reliable-Rpc-Communication` theory. The second clause (under-braced part of the formula) describes behavior in the prsence of failure.

```
THEORY Realistic-Reliable-Rpc-Communication

include Base-Architectural-Style
```

```
value Conf : CONF

AXIOM Realistic-Reliable-Rpc
      ∀req : R,
      C, C' : C,
      rid : UID |
          Call(Conf, C, C', req, rid) ⇒
            (◇(UpCall(Conf, C, C', req, rid)∧
            ⊕□¬UpCall(Conf, C, C', req, rid)∧
            ⊖⊡¬UpCall(Conf, C, C', req, rid)∧
            (∃resp : R |
               ◇(ReturnUpCall(Conf, C, C', resp, rid) ⇒
                 ◇ReturnCall(Conf, C, C', resp, rid)))))∨
            (failure ∧ (∃req' : R |
               UpCall(Conf, C', C, req', rid) ∧ failure(req')))
END
```

The architecture providing realistic reliable RPC communication is refined into four middleware architectures. The three of them are concrete and they are based on well-known middleware infrastructures. The first architecture is specific to the EJB middleware infrastructure and consists of an EJB client-side object proxy and an EJB server-side object proxy, which combine functionality of the JAVA RMI broker. The directory that represents the EJB architecture is linked to the corresponding implementation, available in the implementation repository. Similarly, the second concrete architecture consists of a client-side MTS context wrapper and a server-side MTS context wrapper, which combine functionality of the DCOM ORPC broker. The directory that represents this architecture is also linked to the corresponding available implementation. The third architecture consists of a client-side CORBA proxy and a server-side CORBA skeleton, which combine functionality of a CORBA compliant ORB broker. The directory that represents this architecture is linked to more than one implementation (e.g. ORBIX2.1, OMNIORB etc.).

Finally, the fourth middleware architecture that refines the one providing realistic reliable RPC communication, additionally provides transactions. The following theory describes the properties provided by this architecture. Note that, in order to describe transactional properties the following theory includes definitions given in the `Transactional-Style` theory that extends the `Base-Architectural-Style` theory.

```
THEORY Transactional-Rpc-Communication

include Transactional-Style

include Realistic-Reliable-Rpc-Communication
```

AXIOM Transactions:
$\forall t : \mathcal{T} \mid$
$(atomicity(Conf, t) \wedge isolation(Conf, t))$
END

The middleware architecture that provides transactional realistic reliable Rpc communication, is refined into two others which provide respectively nested, and join/split transactional semantics. The latter architecture is not refined into a concrete architecture. On the contrary, the former architecture is finally refined into three concrete middleware architectures. The first one is based on Corba and combines a Corba client proxy, a Corba server skeleton, and and the Ots and Ccs common object services. The second architecture is specific to Ejb and combines an Ejb client-side object proxy, an Ejb server-side object proxy, and the Jts service. Finally, the third architecture is based on the Dcom infrastructure and combines a client-side Mts context wrapper, a server-side Mts context wrapper, and the Mts transaction service.

III.4.2 Middleware retrieval

The requirements of the distributed file system application comprise those for reliable Rpc communication and transactions. Those requirements are given respectively by the `Cmodel` and the `Tmodel` properties as described in Section II.3. Given those requirements, the architect is supposed to refine them into a concrete middleware architecture that satisfies them. With the design repository in place, the architect can first check whether a similar refinement process took place in the past. In other words, he can try to locate paths in the refinement history that may lead to concrete middleware architectures that satisfy the requirements imposed by the application. Before getting to the retrieval algorithm, proposed in Section III.2.2, note that the `Cmodel` requirements were described based on definitions given in the `Base-Architectural-Style` theory and that the `Tmodel` requirements were described using definitions given in the `Transactional-Style` theory which extends the `Base-Architectural-Style` theory. Hence, what is needed from the repository is a middleware architecture whose proporties are described using definition given in the `Transactional-Style` theory, or an extension of that theory. Moreover, the architect has to complete the required properties by applying the completion pattern proposed in Section III.1.2.

Then, the first step of the algorithm, given in Section III.2.2, comprises locating middleware architectures within the design repository, whose properties are described using at least the `Transactional-Style` theory. Starting from the root directory of the design repository, every sub-directory is recursively visited. For every sub-directory it is checked whether the STeP theory file stored in the sub-directory, includes the `Transactional-Style` theory. If the previous condition holds, the recursive step terminates and current directory is returned as a result. Figure III.8 gives the results of performing this step. The directory that contains information for the middleware architecture that provides transactional realistic reliable RPC communication is the only result.

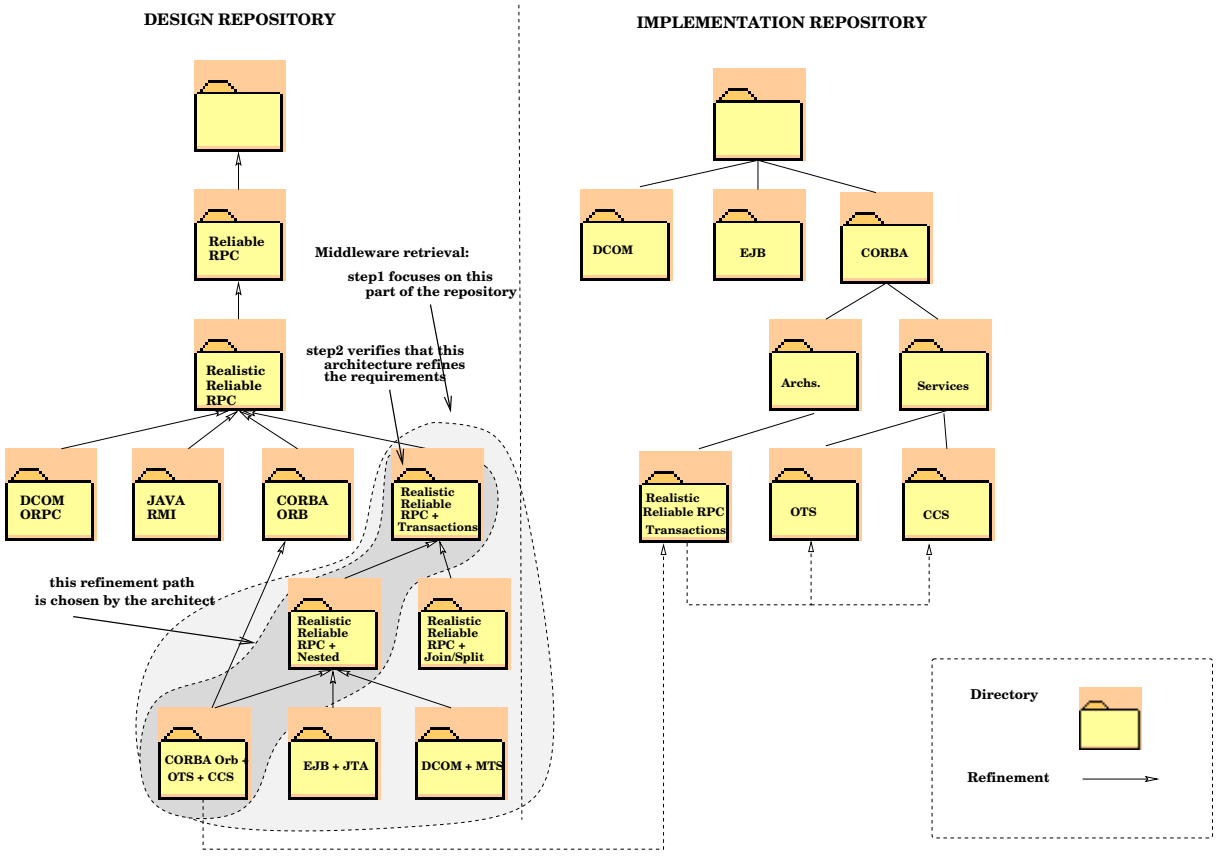

Figure III.8 Case Study: Middleware retrieval

Performing the second step of the algorithm starting from the directory returned from the previous step, proves that `Cmodel` is implied by the `Realistic-Reliable-Rpc` axiom and that the `Tmodel` property is implied by the `Transactions` axiom. Hence, the middleware architecture located in the previous step, actually refines the application requirements. This middleware architecture is gradually refined into three concrete middleware architecture based respectively on EJB, CORBA, and DCOM. Suppose, now, that the architect decides to follow the refinement path that leads to the CORBA based architecture and that this architecture is handed to the designer. The designer can immediately see that the

concrete middleware architecture is linked with a corresponding implementation. Then, his work is to check whether the existing implementation meets his requirements regarding performance, scalability etc. If, however, this is not the case, the designer will start searching into the repository for the architectural elements that make up this architecture. More specifically, the designer will search for a CORBA OTS and a CORBA CCS service based on the standard specification of their properties.

III.4.3 Middleware integration

Following the example scenario, suppose that the concrete middleware architecture resulting from the previous step is the one shown in Figure III.9.

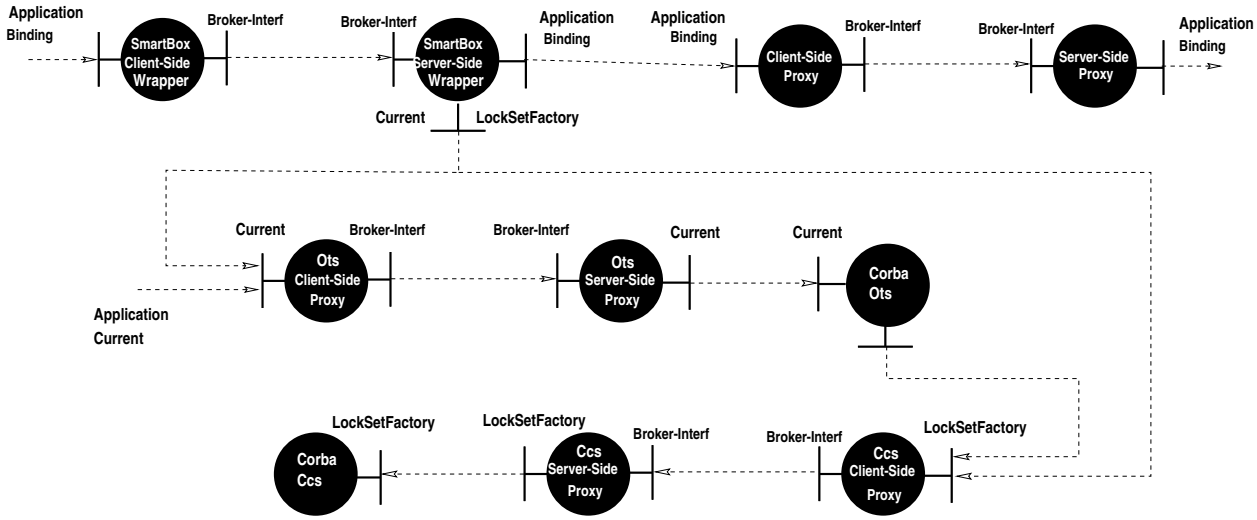

Figure III.9 Case Study: Retrieved middleware architecture

In particular, the resulting middleware architecture comprises a client ORB proxy, a server ORB skeleton, playing, respectively, the role of the client and the server side proxy. In order to complete the middleware implementation the developer has to produce input to the CORBA IDL compiler, which is the tool responsible for generating the implementation of the client and the server side proxies. The input to the CORBA IDL compiler, comprises a CORBA IDL description for every different component type defined in the ADL description of distributed file system application. This input can be produced automatically, by the code generator that supports the ADL proposed in this thesis, given the architectural information coming from the ADL description of the distributed file system application, and the process description given in Figure III.10. This development process instructs the code generator to generate a code that defines a CORBA module, for every different component type defined in the ADL description of the distributed file system application (see lines (1-3)). Moreover, the development process, instructs the code generator to generate a code that defines a CORBA IDL interface, for every interface provided by a component type defined in

```
(1) // Corba module definition
(2) $(Iterate $(component) component
(3)  module $(component.name) {
(4)  // module body
(5)  $(Iterate $(component.provides) provides
(6)        interface $(provides.interface.name) {
(7)          $(Iterate $(provides.interface.operations) operation
(8)            $(Iff $(operation.rettype) $(operation.rettype))
(9)            $(operation.name)($(arglist));
(10)           )
(11)        }
(12)  )
(13)  }
(14))
```

Figure III.10 Developement process directives used to generate input for a CORBA IDL compiler

the ADL description of the distributed file system application (see lines 5-9). Note that the generator is not specific to CORBA. The CORBA specific code is given to the generator as an input and the generator appropriately reproduces this code for every different component type described in the ADL description. Moreover, the retrieved middleware architecture comprises three more components, called `OTS`, `CCS` and `SmartBox`. The OTS and CCS services are explicitly connected through interfaces of type `LockSetFactory`. The reason for this explicit connections is because the `OTS` service is responsible for releasing locks held on objects that participate in a transaction, upon its termination. Since the implementation of the concrete middleware architecture was found in the repository, the proxies that connect the two services are already generated. Hence, the developer's work is trivial and amount in initializing the two services. The `SmartBox` component is the last piece that makes up the retrieved middleware architecture. The functionality provided by this component completes a two-phase locking protocol for a transaction. In particular, the `SmartBox` component acquires a lock on a `Corba` object, at the time when the object begins to participate in a transaction, i.e. at the time when it receives for the first time a request that belongs to context of the transaction. As shown in Figure III.9, the `SmartBox` component is a wrapper to the client-side proxy. A different wrapper for every different component type defined in the ADL description of the application can be generated by the code generator that supports the ADL proposed in this thesis given the architectural information of the distributed file system application and the development process directives shown in Figure III.11. Those wrappers provide the same functionality, but different interfaces. The wrapper implementation creates a CCS lock (see lines (26-28)). Moreover, the wrapper implementation calls the wrapped interface (see line (45)). Finally, the wrapper tries to acquire the lock right before calling the wrapped interface, and if this is called for the first time in the context of a particular transaction (see line (39-44)).

```
(1) $(Iterate $(component) component
(2) $(Iterate $(component.provides) provides
(3) // class definition per interface provided by component
(4) class $(provides.interface.name)SmartClassImpl
(5)         : public $(component.name)BOAImpl {
(6)  public:
(7)         $(provides.interface.name)SmartClassImpl(CORBA::Object_ptr *);
(8)         ~$(provides.interface.name)SmartClassImpl(void);
(9)         $(Iterate $(provides.interface.operations) operation
(10)          ($Iff $(operation.rettype) $(rettype))$(operation.name)($(arglist));
(11)         )
(12)  private:
(13)        // wrapped CORBA object
(14)        CORBA::Object_ptr *p;
(15)        // lock set factory
(16)        CosConcurrencyControl::LockSetFactory_ptr lfactory_ptr;
(17)        // transactional lockset
(18)        CosConcurrencyControl::TransactionalLockSet_ptr lset_ptr;
(19)        // Ots server TM
(20)        CosTransactions::ServerManager_ptr vClientMgr;
(21) }
(22) // Constructor/destructor code
(23) $(provides.interface.name)SmartClassImpl::
(24) $(provides.interface.name)SmartClassImpl(CORBA::Object_ptr *) {
(25)        p = target;
(26)        // initiate a transactional lock set
(27)        lfactory_ptr = _init();
(28)        lset_ptr = lfactory_ptr->create_transactional();
(29)        vServerMgr = CosTransactions::ServerManager::IT_create ();}
(30) // class implementation per interface provided by component
(31) $(provides.interface.name)SmartClassImpl::
(32) $(Iterate $(provides.interface.operations) operation
(33) $(Iff $(operation.rettype) $(operation.rettype))$(operation.name)($(arglist)()) {
(34)        CosTransactions::Control_ptr t_control_ptr;
(35)        CosTransactions::Coordinator_ptr t_coordinator_ptr;
(36)        CosTransactions::Current_ptr t_current_ptr;
(37)        // get current
(38)        t_current_ptr = vServerMgr->create_current();
(39)        if (t_current_ptr) {
(40)           t_control_ptr =  t_current_ptr->get_control();
(41)           t_coordinator_ptr = t_control_ptr->get_coordinator();
(42)        // try to acquire transactional lock
(43)           lset_ptr->lock(t_coordinator_ptr, write);
(44)        // if lock acquired make the actual call to the wrapped object
(45)        $(Iff $(operation.rettype)) return) p->$(operation) ($(operation.arglist));}))))
```

Figure III.11 Developement process directives used to generate `SmartBox` wrapper implementations

III.5 ADL Evaluation

This chapter discussed the ADL support for the systematic middleware synthesis. The proposed ADL support aims at helping, the architect and the designer when refining abstract requirements of an application into a concrete middleware architecture, and the developer when implementing the concrete middleware architecture and integrating it within an application. This section evaluates the ADL support, proposed for the systematic synthesis of middleware, regarding the issues of refinement and traceability.

As mentioned in [56], an ADL that aims at helping the software development process should enable the correct refinement of abstract requirements into a concrete executable software system. Among the ADLs examined in the context of this thesis only SADL, RAPIDE and ASTER support refinement. In SADL, refinement mappings are used to verify the correctness of subsequent refinement steps. Similarly, in RAPIDE, event mappings are used for generating comparative simulations of an architecture at different refinement steps. In ASTER, properties guaranteed by a middleware architecture can be refined into stronger and more detailed ones. Moreover, the ASTER ADL, is supported by a repository, which is used to store properties. The repository structure reflects the refinement relation among the properties. The refinement structure can be used be the architect to trace changes across different levels of refinement with respect to the strength of the properties provided by available middleware architectures.

This chapter proposed some necessary improvements to the approach previously suggested by ASTER. First of all, except for the properties guaranteed by a middleware architecture, a textual description that describes the elements that make up this architecture is also stored in the repository. Furthermore, the repository contains properties describing the observable behavior of the individual elements that make up a middleware architecture. The combination of those two features allows to better trace how abstract middleware architectures are refined into concrete ones. More precisely, it allows to trace how elements of an abstract middleware architecture are decomposed into more concrete elements, and further enables to verify, based on the properties describing the behavior of the individual elements, that their composition is correct with respect to the properties guaranteed by the basic middleware architecture. Finally, the last improvement proposed in this thesis relates to the navigation within the middleware repository structure. So far in ASTER navigation was done only with the help of a theorem prover. This thesis proposed a simple and more efficient method which reduces the use of the theorem prover, when trying to locate middleware architectures that refine requirements of an application. The basic idea is to try to focus inside the repository, on middleware architectures that satisfy requirements for a particular architectural style required by the application. The previous task is achieved without using a theorem prover. Then, it is checked whether the architectures located in the previous step are refined to concrete ones, which satisfy the application requirements. This step makes use of the theorem prover.

ASTER is the only one among the examined ADLs that allows to gradually refine an

architecture into an executable system. In particular, ASTER provides a tool that enables building a CORBA implementation of an application, given the application's ADL description [37, 39]. Moreover, there exist ADLs that do not support refinement and traceability but do provide support for constructing an executable system, out of its abstract ADL description. In particular, DCL specifications can be compiled into an executable application built on top of an infrastructure called Actor Foundry. This translation process is based on the work presented in [82]. Similar translation processes could be devised for deriving CORBA, EJB, DCOM implementations of a DCL specification. DURRA comes along with a middleware infrastructure consisting of a set of channels and cluster implementations providing certain properties. It further supports the automatic construction, deployment and management of an application on top of this middleware infrastructure. In the same spirit, C2, comes along with a middleware infrastructure that can be customized according to the needs of the application. Tools for constructing an application configuration based on this infrastructure are also provided. Support for constructing an executable system is also provided by UNICON. The set of possible connectors is fixed, and depending on their kind, different strategies are used to integrate them within an application. DARWIN, provides a tool that is very similar to the one previously provided by ASTER, for building a CORBA implementation of an application, given the application's ADL description.

One common point in the various tools that come along with the aforementioned ADLs is that they are specific to a middleware infrastructure (e.g. Actor Foundry), or specific to families of infrastructures that conform to a well-known standard (e.g. CORBA). Instead of providing infrastructure-specific tools, this thesis proposed a method for building an executable system, based on the ADL description of an application, that does not depend on any particular infrastructure. Moreover, the proposed method allows exploiting, as necessary, the use of tools that are specific to available middleware infrastructures. This method comprises providing development process directives along with every concrete middleware architecture stored in the implementation repository. Those directives are specific to the particular middleware architecture and describe the way to complete its implementation with the application dependent parts that are missing. Then, an infrastructure independent code generator reuses the provided development process directives for generating the code that completes the middleware implementation. The proposed approach reflects the main purpose of an ADL. An ADL is supposed to provide generic methods and tools that ease the software development process, promote software reuse, and reduce the cost to exploiting and evaluating different available solutions to a particular problem.

IV Middleware Exchange

The systematic middleware exchange process comprises: (1) synthesizing a new middleware architecture that reflects changes in the application requirements or in the availability of middleware, (2) exchanging the old middleware architecture with the new one while preserving the application consistency. Support for the systematic synthesis of middleware presented in the previous chapter can be used to synthesize a new middleware architecture that satisfies the application's evolving requirements, or reflects changes in the availability of middleware. Hence, what remains to be examined is the strategy to be used for exchanging the old middleware with the new one.

This chapter proposes a strategy for exchanging middleware. First, the requirements for correct middleware exchange are identified, based on previous work on dynamic reconfiguration. Those requirements comprise: preserving the consistency of the application, minimizing the disruption introduced in the execution of the application, and guaranteeing that the exchange will be reachable, meaning that it will take place within a finite time. Then, the above requirements are precisely defined based on the specification model introduced in Section II.2.4. Finally, a concrete middleware architecture that satisfies those requirements is designed.

IV.1 Middleware Adaptation

As already mentioned in Section I.3, the requirements of an application and the availability of middleware may change during the lifetime of the application. Consequently, the middleware synthesized for the particular application should adapt so as to reflect those changes. In general, the requirements of an application may strengthen, i.e. the new requirements imply the old ones. Take for instance, the case of an application, executing in a Local Area Network (Lan). Suppose that eventually the application gets connected to the Internet. Typically, the application will additionally require for communication to be secure. Hence, the new requirements are stronger than the old ones. The requirements of an application may also weaken, i.e. the old requirements imply the new ones. Take for example the case of an application that requires a fault tolerant logging mechanism. Suppose that eventually a Raid storage system is installed. Then, fault tolerance is no longer needed and consequently the new requirements of the application are weaker compared to

the old ones. Finally, the requirements of an application may also change arbitrarily, meaning that some requirements strengthen, while others weaken. An example that justifies this statement could be the combination of the two aforementioned examples.

Changes in the availability of the infrastructure, used for building a middleware architecture, may also trigger the need to adapt a middleware. For example, in practice we frequently upgrade middleware infrastructures with later versions. Then, a middleware that relies on the infrastructure should also be upgraded so as to make use of the latest version. Adapting a middleware to reflect arbitrary changes in the application requirements or the availability of the underlying infrastructure requires, *in the worst case*, performing major changes to its architectural structure. Hence, from now on it is assumed that the middleware adaptation amounts in exchanging a middleware architecture with a new middleware architecture. The new middleware architecture can be built using the systematic middleware synthesis process detailed in Chapter III.

From an architectural point of view, the overall middleware exchange process is nothing but a special case of changing the configuration of an application. The basic goal is to exchange a middleware connector for another one that preserves the current requirements of the application and reflects changes in the availability of the underlying infrastructure. The remainder of this section identifies the basic requirements for correct middleware exchange based on previous work done in the field of dynamic reconfiguration.

In [44], the authors discuss about the basic properties characterizing a process that systematically manages changes, i.e. addition, removal, substitution of components, in the configuration of a running application. The first of the essential properties, mentioned by Kramer and Magee, is the one of preserving the correct execution of the application. As said in the paper, "*(configuration) changes should leave the system in a consistent state*". A consistent state is one from which the application can continue normally rather than progressing towards an error state.

The second essential property is the one of introducing minimal disruption while changing the configuration of the application. As mentioned in the paper, "*(configuration) changes should minimize the disruption of the system*". In other words, it should not be necessary to suspend the execution of the whole application so as to change a part of it.

At this point it worths noticing that in case of real-time, critical applications, disrupting the execution of the application immediately implies violating the application consistency. In general, for a wide variety of applications, efficiency is a primary concern. Consequently, the set of correctness criteria that define consistency for an application includes constraints regarding the efficient execution of the application. Based on the previous remark, the first essential property for correct middleware exchange is *to preserve application consistency, i.e. the middleware exchange must leave the application in a correct state, and it must introduce minimal disruption in the execution of the application.*

A third essential property identified in [44] is *reachability* of changes, meaning that a configuration change must take place within finite time. In analogy to the previous, the

second essential property that should be preserved by a systematic middleware exchange process is that *the middleware exchange will be reachable, i.e. the middleware exchange will take place within finite time.*

Let us now take a closer look at the change strategy proposed by Kramer and Magee given the requirements identified in [44]. According to the authors, the process that manages changes should be capable to identify a minimal set of architectural elements *affected* by the change. Components of the application that do not belong to the set of affected components should be able to operate normally, as if there was no configuration change in progress. More specifically, in the case of component substitution, the set of components affected by the change comprises all components that can possibly initiate requests to the component that is to be substituted. To substitute a component of an application the following steps are performed by the change management process: (1) Block all requests, coming from components affected by the change, targeted to the component that is to be substituted; (2) Wait until the time when the component that is to be substituted is not engaged in servicing any pending requests, nor in any request that it initiated; (3) Remove all links to and from the component that is to be substituted; (4) Remove the old component; (5) Create a new component; (6) Set up the communication links as prescribed by the architectural description of the application; (7) Unblock all requests that were blocked during the first step;

In analogy to the previous protocol for substituting application components, the following are the derived steps for exchanging a middleware architecture with another one: (1) Block all requests, coming from components affected by the change, targeted to the middleware; (2) Wait until the time when the middleware that is to be substituted is not engaged in servicing any pending requests, nor any request that it initiated; (3) Remove all links to and from the middleware; (4) Install a new middleware; (5) Un-install the old middleware; (6) Set up the communication links as prescribed by the architectural description of the application; (7) Unblock all requests that were blocked during the first step; It is immediately visible that the aforementioned protocol is not very helpful for exchanging middleware systematically. More specifically, the previous protocol does leave the application in a correct state after the exchange. On the other hand, it does not minimize application disruption. Indeed, the set of components affected by the middleware exchange comprises components of the application that interact through the middleware. In the most common case, all of the application components interact through the same middleware connector. Consequently, the whole application is affected by the middleware exchange, and there is no point of trying to locate a minimal set of affected components for optimizing the previous change protocol [23]. Hence, blocking only requests coming from the affected components, becomes a useless optimization. In principle, the previous idea was based on the fact that the application could possibly afford a few blocked requests given that the set of affected components is a minority of the overall set of components that make up the configuration of the application. However, the more the set of affected components grows, the less tolerable becomes the fact of blocking requests. In the case of exchanging a middleware with another one, practically all requests to and from the

middleware should be blocked. Redirecting requests could have been a good solution for making configuration changes while not disrupting the application. In [62], the authors suggest an approach for component replacement, based on the redirection of requests, that were originally targeted to the old server, to the new server. The particular replacement protocol is performed by a configuration management process, called controller, which is responsible of enforcing predefined constraints that guarantee the correct execution of the application. This solution seems, however, to have little chance to work out in the case of middleware exchange. One typical problem is that part of the middleware implementation is loaded in the application components' address space. Then, keeping both middleware operational at the same time, implies using part of the code of both middleware in the same address space, which is difficult if their implementations conflict.

Reaching a dead-end so far, and since it is not practically possible to minimize application disruption during the middleware exchange process, what is left as an option is to try minimizing the overall duration of the middleware exchange process i.e. *whatever disruption is introduced by the middleware exchange process, it does not last for long.* The duration of the middleware exchange process includes the time it takes for the middleware to reach a state where it is safe to perform the exchange. According to Kramer and Magee, this is a state where there is no request that is pending. From now on we call such a state an *idle state.* If we assume, however, that a *safe state* is any state where pending requests do exist but state information related to them can be transfered from the old to the new middleware, which is further capable of bringing pending requests into completion, then the duration of the exchange process can be decreased. For example, if any state of the old middleware can be mapped into a state of the new middleware, the duration of the exchange process drops to zero. Several approaches, dealing with the substitution of application components, improved the approach proposed by Kramer and Magee, by following the aforementioned technique, involving state transfer [25, 89].

Summarizing, *correct middleware exchange must take place at a time when middleware is in an idle state, or if possible at a time when middleware is in a safe state.* Given the previous informal definition of safe state, mechanisms that detect whether a middleware is in a safe state and mechanisms that map this state into a state of the new middleware depend on a particular exchange situation. Hence, building a middleware that can be exchanged while preserving application consistency requires knowing, in advance, the precise exchange situation. This, however, is a limitation that should be avoided in the systematic middleware exchange process. As discussed at the beginning of this section, both the requirements of the application and the availability of the middleware can change in any arbitrary way. Hence, there is no way to a priori know for sure, or even speculate, the precise middleware exchange situations that may eventually arise during the lifetime of the application. Consequently, it is not possible to a priori know which state is safe for exchange and how to detect it. In order to cover this limitation, and be able to deal with arbitrary changes in the application requirements and the availability of middleware, while preserving application consistency, *support that detects whether it is safe to exchange a middleware with another one should adapt regarding a particular exchange situation.*

Regarding *reachability* of changes, Kramer and Magee propose constraining the behavior of the application so as to make the change possible within a finite time. More specifically, in [44], the authors assume that interactions between application components complete within finite time. Interactions are realized within transactions, which are are committed or aborted in a finite time. Hence, during the exchange of a transactional middleware with another one, the initiation of new transactions is blocked and previously initiated transactions complete in finite time. Consequently, the exchange is always reachable. However, constraining the behavior of the application is not always possible, and certainly it is not desirable if it can be avoided. One alternative option, so as to avoid constraining the behavior of the application when exchanging a middleware for another one, is to enforce an interaction that leads middleware to a safe state.

During the execution of the application, the state of the middleware changes as a result of interactions between the middleware and the components of the application. For instance, application components can initiate and terminate transactions, or security sessions by calling specific operations provided by the underlying middleware. Moreover, the state of the middleware can change as a result of events that do not relate with the application lying on top of the middleware. For example, network timeouts may cause the middleware to re-transmit requests issued by components of the application. Finally, the state of the middleware may change due to interaction between services that constitute the middleware. Hence, enforcing a safe state requires a process that introduces an interaction, which drives middleware to safe state. Such a process may behave as follows:

- Play the role of the application, by issuing requests that cause the middleware to reach a safe state. For example, issuing application requests for aborting all pending transactions.
- Play the role of the middleware, by generating events that make the middleware look as if it is in a safe state. For example, raising exceptions, which notify the application that all pending transactions are rolled-back.
- Play the role of the software or hardware that the middleware relies on, by generating independent events that cause the middleware to reach a safe state. For example, introducing network and hardware failures, makes the middleware rollback all pending transactions.

If we assume that it is always possible to enforce an interaction that drives a middleware to a safe state, then it is possible to apply this idea for exchanging middleware within a finite time. However, adopting this approach so as to assure reachability conflicts with the first property that must characterize the systematic middleware exchange process. In particular, the systematic middleware exchange process must disturb the application as less as possible. Consider, then, the case of aborting some transactions that were just about to complete at the time of the exchange. Typically, the application will have to retry again all the aborted transactions. Hence, the requirement for minimal disruption is not preserved.

Consider now that the middleware returns an exception for all the requests sent by the application components, which execute a heavy-weight algorithm, or a simulation. Again, in this case, the work has to be repeated. Even worst, enforcing a safe state implies changing the semantics of the application from the standpoint of an outside observer. For example, a user-initiated transaction is typically aborted either by the user, or by the application itself in case where some failure occurs. Hence, if well-formed transactions are aborted due to an enforced interaction, the user gets the impression that the application is faulty, or that the underlying middleware is not correct, or finally that the hardware, or software where the middleware relies on is not correct.

Hence, trying to actively drive middleware into a safe state is a rather dangerous approach, the only option that is left is to passively wait for the middleware to reach a safe state. Moreover, waiting for the middleware to reach the safe state is acceptable only if the waiting time does not exceed reasonable limits. Guaranteeing reachability of safe state, within a finite time is certainly not possible without constraining the behavior of the application. Making, however, long waits less likely can be done by selectively blocking the components that interact through the middleware. The purpose of blocking is to prevent components of the application from issuing requests that keep the middleware away from a safe state. For example, preventing the application components from issuing requests that initiate new transactions is a way to assist the middleware in reaching a safe state. Similarly, preventing the application components from opening new security sessions would help the middleware to smoothly slip into a safe state.

The requirements introduced in this section were described in an informal manner. In the next section those properties are further analyzed and formally specified, based on the specification model given in Section II.2.4.

IV.2 Requirements Specification and Analysis

The basic requirements for correct middleware exchange are: to preserve application consistency and to assist in reaching a safe state. To formally describe the previous properties there is a need to properly extend the `Base-Architectural-Style` theory. The resulting extended theory, called `Exchangeable-Middleware-Style`, is not explicitly defined in the document. Instead, the required extensions are introduced gradually, in the following sections, during the analysis and specification of the properties characterizing a correct middleware exchange process.

IV.2.1 Preserving consistency

In order to preserve application consistency during middleware exchange, the exchange must take place at a time when there exist no pending requests, or if possible, at a time when there exist pending requests and the state of the old middleware can be mapped into

a state of the new middleware, which is capable to satisfy requirements regarding pending requests. In order to specify the previous property formally, there is a need to extend the `Base-Architectural-Style` with means that allow: (1) characterizing pending requests, (2) defining state mapping functions, and (3) describing a particular exchange situation.

Hence, the following type is defined in the `Exchangeable-Middleware-Style`, and is used in the specifications to define boolean functions, which accept as arguments a request and a corresponding unique identifier, and return true if a request is pending (i.e. a middleware property is not satisfied for req, rid), and false otherwise. For example, let us consider a middleware that provides *Security*, a property that refines the application's requirements for secure communication. Suppose that according to the *Security* property, a security session s, is a set of requests $s.RXID$ sent and delivered with respect to a particular security policy. A security session can be initiated and terminated by components of the application, by calling operations provided by the middleware. A security session s that is initiated, and not yet terminated is a pending security session. Consequently, for all requests and their corresponding unique identifiers (req, rid) belonging to $s.RXID$, $Pending_{Security}() : \mathcal{PENDING}$ returns *true*.

$$\texttt{type}\ \mathcal{PENDING}() : \mathcal{R} * \mathcal{UID} \rightarrow \{true, false\}$$

Moreover, the following type is defined in the `Exchangeable-Middleware-Style`, and is used in the specifications to define state mapping functions.

$$\texttt{type}\ \mathcal{MAP}() : \Sigma \rightarrow \Sigma$$

Finally, to provide means for describing a middleware exchange situation the following type is defined in the `Exchangeable-Middleware-Style` theory. $XSituation$ is a record that comprises: a unique identifier that characterizes an exchange situation; the old middleware; the new middleware; a set of functions of type $\mathcal{PENDING}$, each one of which is used to characterize pending requests with respect to a particular property provided by the old middleware; a state mapping function which by default is null; and finally two requests, which are issued at the beginning and the end of a particular exchange situation.

$$\begin{array}{l}\texttt{type}\\ \quad XSituation = \{\\ \quad\quad xid : \mathcal{UID},\\ \quad\quad MdwOld : \mathcal{CONF},\\ \quad\quad MdwNew : \mathcal{CONF},\\ \quad\quad PendingFunctions : \{Pending_{Property} : \mathcal{PENDING}\} \neq \emptyset,\\ \quad\quad Map : \mathcal{MAP} = \bot,\\ \quad\quad reqXBEG : \mathcal{R},\\ \quad\quad reqXEND : \mathcal{R}\\ \quad\}\end{array}$$

The `Exchangeable-Middleware-Style` further defines two macros called $SafeState$ and $IdleState$. The former macro defines safe state regarding a particular exchange situation. The latter macro defines idle state as described by Kramer and Magee in [44]. Hence, $SafeState$ holds for a particular exchange situation, if the old middleware is currently in a state σ, and this state can be mapped into a state of the new middleware, which is able to eventually satisfy the requirements of all the requests that were pending at the time of the exchange.

$$
\begin{array}{l}
\texttt{macro} \\
SafeState(\sigma : \Sigma, MdwOld : \mathcal{CONF}, MdwNew : \mathcal{CONF}, \\
\qquad\qquad Pending_{Property} : \mathcal{PENDING}, Map : \mathcal{MAP}) = \\
\quad [MdwOld, \sigma] \wedge \\
\quad (\forall req : \mathcal{R}, \\
\quad\; rid : \mathcal{UID} \mid \\
\quad\quad (Pending_{Property}(req, rid) \wedge \Diamond [MdwNew, Map(\sigma)]) \Rightarrow \\
\quad\quad\quad (\Diamond \Box \neg Pending_{Property}(req, rid)))
\end{array}
$$

Similarly, $IdleState$ holds if the old middleware is in a state σ where no requests are pending.

$$
\begin{array}{l}
\texttt{macro} \\
IdleState(\sigma : \Sigma, MdwOld : \mathcal{CONF}, Pending_{Property} : \mathcal{PENDING}) = \\
\quad [MdwOld, \sigma] \wedge\ (\nexists req : \mathcal{R},\ rid : \mathcal{UID} \mid Pending_{Property}(req, rid))
\end{array}
$$

Then, middleware exchange preserves application consistency if the following property holds for every exchange situation.

$$
\begin{array}{l}
\texttt{PROPERTY Consistency} \\
\forall X : XSituation \mid \\
\quad (X.Map \neq \perp \Rightarrow \\
\quad\; (\exists C, C' : \mathcal{C} \mid \\
\quad\quad Call(X.MdwOld, C, C', X.reqXEND, X.xid)) \Rightarrow \\
\quad\; (\exists \sigma : \Sigma \mid \\
\quad\; \Diamond\!\!\!-(\forall Pending_{Property} : \mathcal{PENDING} \mid \\
\quad\;\; Pending_{Property} \in X.PendingFunctions \wedge \\
\quad\;\; SafeState(\sigma, X.MdwOld, X.MdwNew, Pending_{Property}, X.Map)))) \wedge \\
\quad (X.Map = \perp \Rightarrow \\
\quad\; (\exists C, C' : \mathcal{C} \mid \\
\quad\quad Call(X.MdwOld, C, C', X.reqXEND, X.xid)) \Rightarrow \\
\quad\; (\exists \sigma : \Sigma \mid \\
\quad\; \Diamond\!\!\!-(\forall Pending_{Property} : \mathcal{PENDING} \mid \\
\quad\;\; Pending_{Property} \in X.PendingFunctions \wedge \\
\quad\;\; IdleState(\sigma, X.MdwOld, Pending_{Property}))))
\end{array}
$$

The above property states that if there exists a state mapping function for a particular exchange situation, the exchange can take place at a time when there may exist requests that are pending, otherwise the exchange takes place at a time when no requests are pending, as suggested by the traditional approach proposed by Kramer and Magee in [44].

IV.2.2 Reaching a safe state of the middleware

As already discussed in Section IV.1, reaching a safe state of the middleware within a finite time can not be guaranteed without imposing constraints on the behavior of the application. Hence, a more pragmatic requirement for the middleware exchange process is to assist in reaching a safe state. The previous can be achieved by selectively blocking components of the application, so as to prevent them from initiating requests that drive middleware far from a safe state. For example, among the components of an application that interact through a transactional middleware, the only ones that must remain unblocked are those that are necessary so as to eventually complete a pending transaction. Similarly, among components of an application that interact through a middleware that provides secure communication, the only ones that must remain unblocked are those needed so as to eventually close a security session.

At this point, let us further analyze the previous requirement. More specifically, it should be noticed that improving the blocking granularity, makes the blocking process much more effective. By definition, a component is a unit of data or computation. This definition is quite general and does not reflect details regarding the way a computation is performed by a particular component. For instance, the previous definition does not reflect details regarding whether a component is multi-threaded, or not. If a component is multi-threaded and it is qualified as necessary for driving middleware to a safe state, all of its threads will remain active. Typically, however, only one, or few of those threads are needed so as to drive middleware to a safe state. Consider for instance a multi-threaded component, which initiated a pending transaction. Suppose that it is the same component that completes the transaction, and in particular a thread that executes within the component. In that case only this thread needs to remain active. The rest of the threads, running within the component should be blocked, since they actually keep middleware away from safe state, either by issuing requests through the middleware, or of by steeling CPU cycles from the thread that is needed to drive middleware into a safe state. Summarizing so far, the way computation is performed is essential for improving the blocking granularity, and consequently for making the blocking process more effective.

Based on the remarks so far, the blocking process must selectively block threads of the application components so as to prevent them from issuing requests that would keep middleware away from a safe state. Before getting to the specification of the previous property, there are two issues that need to be addressed: (1) extend the `Base-Architectural-Style` so as to include the definition of threads; (2) extend the `Base-Architectural-Style` so as to associate threads with requests sent by those threads. The aforementioned extensions

are included in the `Exchangeable-Middleware-Style` theory.

Defining threads of execution

Now and for the rest of this document, it is assumed that computation is performed, on behalf of a component, by a set of concurrently executing threads. This set contains at least one thread. Threads of a component have no state but instead share and manipulate the state of the component. Threads can be created, destroyed and synchronized using functionality provided by the underlying middleware. Threads can be either active (i.e. performing some computation), or idle. A thread is characterized by a set of variables (e.g. program counter), the values of which are saved every time a thread is deactivated, and restored every time a thread is activated. Those variables are, in fact, part of the state of the component that owns the threads.

In Section II.2.4, we defined Σ to be a type used in the specifications to define component states. Σ, is seen as a set of variables, a subset of which contains information about threads. Let `type` $\mathcal{THR}$, included in the `Exchangeable-Middleware-Style` theory, be the type used in the specifications to define threads. Based on the discussion so far, the following axiom holds :

`AXIOM Threads`
$$\mathcal{THR} \subset \Sigma$$

In addition, the following function is included in the `Exchangeable-Middleware-Style` theory. The following function takes as argument a component state instance and returns the part of the state information that refers to the thread which is active. Note that the previous function is correct only if we assume that software executes on top of a single processor; when executing on top of a multi-processor more than one thread are active and hence the *ActiveThread* would return a set of threads instead of a single one.

`value` $ActiveThread : \Sigma \rightarrow \mathcal{THR}$

As mentioned, threads can be created, destroyed, synchronized, using functionality provided by the underlying middleware. When two threads t, t' are synchronized, and until the time when the synchronization point is reached, the execution of the threads is dependent. For instance, if t waits for a signal from t', it is possible that there exist requests that will be sent by t only after t gets the signal from t'. Hence, if t' does not reach the synchronization point where it sends the signal to t, t will not be able to continue executing after the synchronization point. Consider now the case of two threads t, t', of the same component C synchronized into a barrier. If one of those threads does not reach the barrier point, neither of them will continue executing after the barrier point. Then until the time that the barrier is reached, the execution of t depends on the execution of t'. Similarly,

the execution of t' depends on the execution of t. Based on the above remarks the following function is included in the `Exchangeable-Middleware-Style` so as to enable describing dependencies among threads. The function takes as arguments two threads t, t' and returns true while the execution of t depends on the execution of t', and false elsewhere.

$$\texttt{value}\ dep : \mathcal{THR} * \mathcal{THR} \rightarrow \{true, false\}$$

In the case of the threads synchronized into the barrier, and until the time the barrier is reached, both $dep(t, t')$ and $dep(t', t)$ return $true$.

Associating threads of execution with requests

In order to associate threads of execution with requests sent or received by those threads, there is a need to redefine the basic *Call*, *UpCall*, *ReturnCall*, *ReturnUpCall* macros that were defined in Section II.2.4. Before giving the new definitions of those macros, the following functions need to be defined:

- The *CallThread* function is used, hereafter, in the specifications, to associate requests exported, at some time by an architectural element, with the thread that actually issued the request. More specifically, the *CallThread* function accepts as arguments a request $req : \mathcal{R}$ and a corresponding unique identifier $rid : \mathcal{UID}$, and returns a thread $t : \mathcal{THR}$ that issued the request.

$$CallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

- Similarly, the *UpCallThread* function is used, hereafter, in the specifications, to associate requests delivered, at some time to an architectural element, with the thread that took in charge of serving the request. More specifically, the *UpCallThread* function accepts as arguments a request $req : \mathcal{R}$ and a corresponding unique identifier $rid : \mathcal{UID}$, and returns an associated thread $t : \mathcal{THR}$.

$$UpCallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

- The *ReturnCallThread* function is used, hereafter, in the specifications, to associate responses to requests exported by an architectural element, with the thread that actually received the response. More specifically, the *ReturnCallThread* function accepts as arguments a response $resp : \mathcal{R}$ and a corresponding unique identifier $rid : \mathcal{UID}$, and returns the associated thread $t : \mathcal{THR}$ that received the response.

$$ReturnCallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

- Similarly, the *ReturnUpCallThread* function is used, hereafter, in the specifications, to associate responses sent, at some time by an architectural element, with the thread that sent the response. More specifically, the *ReturnUpCallThread* function accepts as arguments a response $resp : \mathcal{R}$ and a corresponding unique identifier $rid : \mathcal{UID}$, and returns an associated thread $t : \mathcal{THR}$.

$$ReturnUpCallThread() : \mathcal{R} * \mathcal{UID} \rightarrow \mathcal{THR}$$

Then, the basic macros that describe interaction among the elements of an architecture are redefined as follows:

- The definition of the *Call* macro is as before with only one additional clause. This clause associates a request req, exported by a component C towards a component C', with the thread that actually issued the request. This is done by setting the return value of the $CallThread(req, rid)$ function to be equal with the return value of the $ActiveThread(\sigma)$ function.

$$\begin{array}{l}
\texttt{macro} \\
\quad Call(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad \exists o : \mathcal{O}_r, \\
\qquad ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad ((ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in ir) \wedge \\
\qquad\quad (req \in rang(o)) \wedge \ominus[C, \sigma] \wedge req = o(\sigma, rid) \wedge \\
\qquad\quad CallThread(req, rid) = ActiveThread(\sigma))
\end{array}$$

- Similarly, the definition of the *UpCall* macro is as before with only one additional clause. This clause associates a request req, imported by a component C', with the thread that takes in charge of serving the request. This is done by setting the return value of the $UpCallThread(req, rid)$ function to be equal with the return value of the $ActiveThread(\sigma)$ function.

$$
\begin{array}{l}
\texttt{macro} \\
\quad UpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, req : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad (C, C' \in Conf) \wedge \\
\qquad (\exists ir : \mathcal{I}_r, \\
\qquad ip : \mathcal{I}_p, \\
\qquad o : \mathcal{O}_p, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad (ir \in \sharp(I_r)C) \wedge \\
\qquad\quad (ip \in \sharp(I_p)C') \wedge \\
\qquad\quad (ir \rightsquigarrow ip) \wedge \\
\qquad\quad (o \in \sharp(I_p)C') \wedge \\
\qquad\quad ((\sigma, req, rid) \in dom(o)) \wedge \\
\qquad\quad \Diamond Call(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge [C', o(\sigma, req, rid)] \wedge \\
\qquad\quad UpCallThread(req, rid) = ActiveThread(\sigma))
\end{array}
$$

- In the same spirit, the definition of the *ReturnCall* macro is enhanced with an additional clause that assigns to $ReturnCallThread(req, rid)$ the value of the $ActiveThread(\sigma)$ function.

$$
\begin{array}{l}
\texttt{macro} \\
\quad ReturnCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad \exists \sigma : \Sigma \mid \\
\qquad\quad \Diamond ReturnUpCall(Conf, C, C', resp, rid) \wedge \ominus[C, \sigma] \wedge \\
\qquad\quad [C, response(\sigma, resp, rid)] \wedge\ ReturnCallThread(req, rid) = ActiveThread(\sigma)
\end{array}
$$

- Finally, the definition of the *ReturnUpCall* macro is also extended with an additional clause that assigns to $ReturnUpCallThread(req, rid)$ the value of the $ActiveThread(\sigma)$ function.

$$
\begin{array}{l}
\texttt{macro} \\
\quad ReturnUpCall(Conf : \mathcal{CONF}, C : \mathcal{C}, C' : \mathcal{C}, resp : \mathcal{R}, rid : \mathcal{UID}) = \\
\qquad \exists req : \mathcal{R}, \\
\qquad \sigma : \Sigma \mid \\
\qquad\quad \Diamond UpCall(Conf, C, C', req, rid) \wedge \ominus[C', \sigma] \wedge \\
\qquad\quad resp = response(\sigma, rid) \wedge\ ReturnUpCallThread(req, rid) = ActiveThread(\sigma)
\end{array}
$$

Based on the revised definitions included in the `Exchangeable-Middleware-Style` it is now possible to define formally the property that must hold so as to assist in reaching a safe state. The definition of this property makes use of a type, called $\mathcal{SEQ}$, defined in the `Exchangeable-Middleware-Style` theory. $\mathcal{SEQ}$ is an array of requests whose last item is always null ($\perp$), and it is used in the specifications to declare sequences of requests.

Moreover, the $filter$ function, defined in the `Exchangeable-Middleware-Style` theory, is a function that takes as argument a sequence of requests seq and a thread t, and returns a sequence of requests which is the same as seq but the requests sent by the thread t and the requests sent by threads that depend on t.

```
type
SEQ = array[1..] of R * UID

value filter() : SEQ * THR → SEQ

AXIOM FiniteSequence:
∀seq : SEQ |
      (∃i : int |
         i ≥ 1∧
         (∀j : int | j ≥ i ∧ seq[j] = ⊥)∧
         (∀k : int | k < i ∧ seq[k] ≠ ⊥))

AXIOM Filtering:
∀seq, seq′ : SEQ,
t : THR |
      seq′ = filter(seq, t) ⇒
(∀i : int |
         (i ≥ 1 ∧ seq[i] ≠ ⊥ ∧ CallThread(seq[i]) ≠ t ∧ ¬dep(CallThread(seq[i]), t)) ⇒
            (∃j : int | j ≥ 1 ∧ seq′[j] ≠ ⊥ ∧ seq′[j] = seq[i]))∧
(∀i : int |
         (i ≥ 1 ∧ seq′[i] ≠ ⊥) ⇒
            (∃j : int | j ≥ 1 ∧ seq[j] ≠ ⊥ ∧ ¬dep(CallThread(seq[j]), t) ∧ seq′[i] = seq[j]))

macro
CallSEQ(Conf : CONF, seq : SEQ) =
     (∃C, C′ : C |
        Call(Conf, C, C′, ♯(R)seq[1], ♯(UID)seq[1]))∧
        (∀i, j : int |
           i ≥ 1 ∧ seq[i] ≠ ⊥ ∧ j ≥ 1 ∧ seq[j] ≠ ⊥ ∧ j > i∧
           (∃Ci, C′i, Cj, C′j : C |
              ◇(Call(Conf, Ci, C′i, ♯(R)seq[i], ♯(UID)seq[i]) ⇒
                 ◇Call(Conf, Cj, C′j, ♯(R)seq[j], ♯(UID)seq[j])))))
```

Finally, the macro $UnNecessary$ holds for a given thread t if t can be blocked during the middleware exchange process, i.e. if it is not needed so as to drive middleware to safe

state.

```
macro
```
$$
\begin{array}{l}
UnNecessary(Conf : \mathcal{CONF}, t : \mathcal{THR}, X : XSituation) \ = \\
\quad \forall seq : \mathcal{SEQ} \mid \\
\quad\quad CallSEQ(Conf, seq) \Rightarrow \\
\quad\quad\quad \Diamond(\exists \sigma : \Sigma \mid \\
\quad\quad\quad\quad \forall Pending_{Property} : \mathcal{PENDING} \mid \\
\quad\quad\quad\quad\quad Pending_{Property} \in X.PendingFunctions \wedge \\
\quad\quad\quad\quad\quad SafeState(\sigma, X.MdwOld, X.MdwNew, Pending_{Property}, X.Map))) \Rightarrow \\
\quad\quad \exists seq' : \mathcal{SEQ} \mid \\
\quad\quad\quad seq' = filtered(seq, t) \wedge \\
\quad\quad\quad\quad (CallSEQ(Conf, seq') \Rightarrow \\
\quad\quad\quad \Diamond(\exists \sigma : \Sigma \mid \\
\quad\quad\quad\quad \forall Pending_{Property} : \mathcal{PENDING} \mid \\
\quad\quad\quad\quad\quad Pending_{Property} \in X.PendingFunctions \wedge \\
\quad\quad\quad\quad\quad SafeState(\sigma, X.MdwOld, X.MdwNew, Pending_{Property}, X.Map)))
\end{array}
$$

Then, the requirement for the middleware exchange is that only threads that are necessary for driving middleware into a safe state are allowed to issue requests through the middleware, for the duration of the middleware exchange. The above is given formally by the `Reachability` property that follows.

```
PROPERTY Reachability
```
$$
\begin{array}{l}
\forall X : XSituation, \\
req : \mathcal{R}, \\
rid : \mathcal{UID} \mid \\
\quad \exists C, C', C'', C''' : \mathcal{C} \mid \\
\quad\quad Call(X.MdwOld, C, C', X.reqXBEG, X.xid) \wedge \\
\quad\quad \Diamond(Call(Conf, C'', C''', req, rid) \wedge \\
\quad\quad \Diamond Call(X.MdwOld, C, C', X.reqXEND, X.xid))) \Rightarrow \\
\quad\quad \neg UnNecessary(Conf, CallTread(req, rid), X)
\end{array}
$$

IV.3 Middleware Architecture Design

Given the requirements for a systematic middleware exchange process, described in the previous section, this section details the general design of a concrete exchangeable middleware architecture that satisfies those requirements. This general design was first introduced in [86]. In particular, except for the basic architectural elements used to design

a middleware architecture that mediates the interaction between application components, an exchangeable middleware architecture must further comprise architectural elements that are able to:

- Detect whether, or not the middleware is in a safe state. As detailed in Section IV.1, in order to support arbitrary changes in the application requirements and the availability of middleware, the architectural elements that detect whether it is safe to exchange a middleware, must adapt regarding a particular exchange situation. Consequently, those architectural elements must be *exchangeable*.

- Block threads of execution that are not needed for driving middleware into a safe state. The architectural elements, responsible for the blocking process do not depend on the exchange situation and, thus, this part of the middleware architecture is *static*.

- Perform the exchange, i.e. load the new middleware in place of the old one. The architectural element that actually performs the exchange, is called *coordinator*, and is also in charge of adapting the elements that detect safe state.

Figure IV.1, gives an overall view of an exchangeable middleware architecture. Each one of the different types of architectural elements, used to build an exchangeable middleware architecture is further detailed in the remainder of this section.

IV.3.1 Detecting safe state

Given the definition of idle state, described by the *IdleState* macro defined in Section IV.2.1, it becomes obvious that detecting an idle state does not depend on any particular exchange situation. In particular, detecting an idle state amounts to monitoring the interaction between the application and the middleware, in a way that allows to reason about whether there exist requests that are pending. For example, a mechanism that detects idle state for a middleware that provides reliable communication, simply monitors requests that were sent and received by the application components, and it reports that the middleware is in an idle state if for every request sent, a response is delivered back to the sender. Similarly, a mechanism that detects idle state for a transactional middleware simply monitors requests, for beginning and completing transactions. This idle state detector reports idle state if all requests for beginning transactions were coupled with the corresponding requests for completing them. Since, idle state detectors are independent of the particular exchange situation, they can be easily reused within different middleware architectures that provide the same properties. In order to further maximize reuse it is assumed that a mechanism detects idle state in a per-property manner. This means that for a middleware architecture that provides two distinct properties like transactional interaction and secure interaction, two different mechanisms are used; one that detects idle state with respect transactions, and one that detects idle state with respect to security.

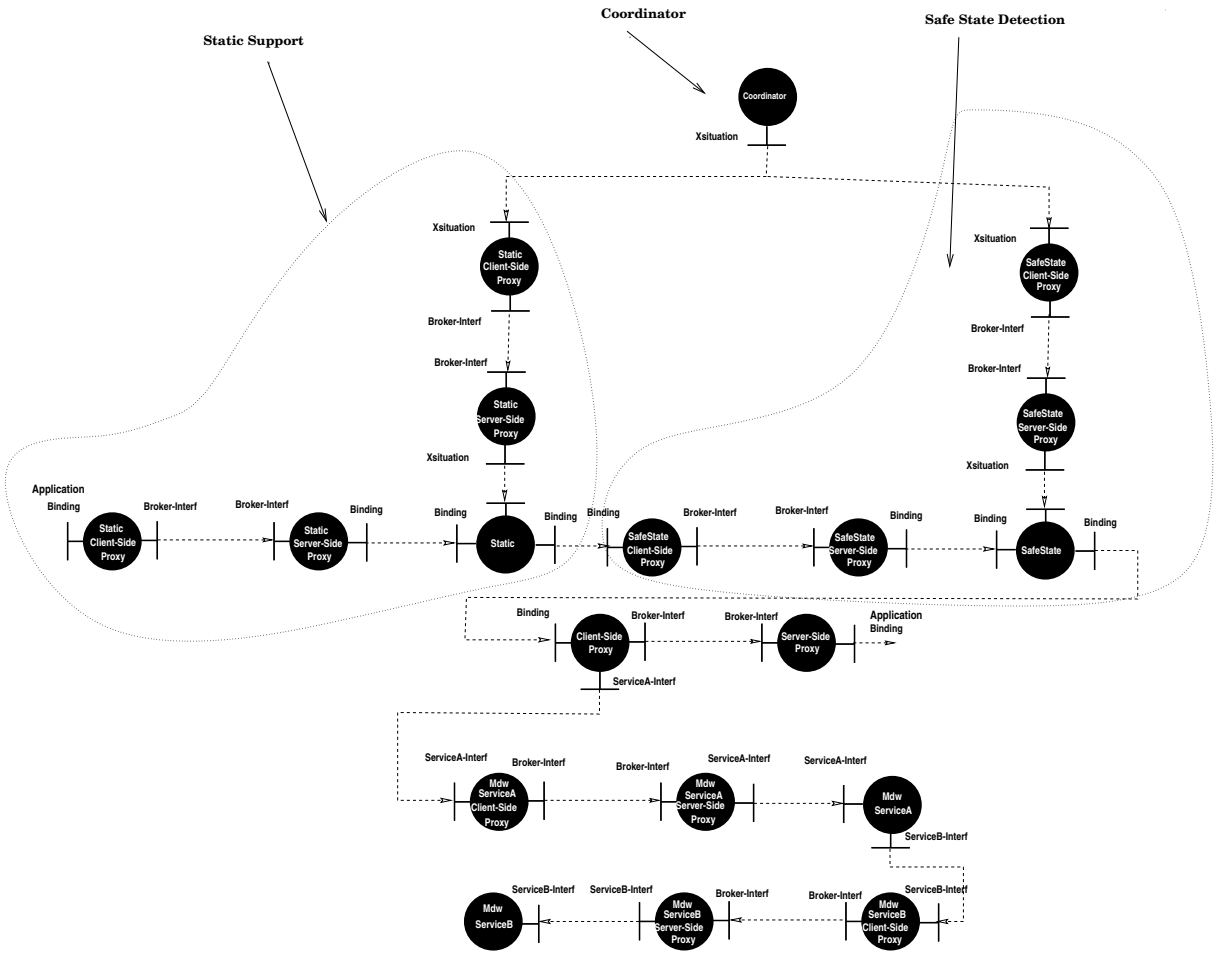

Figure IV.1 An exchangeable middleware architecture

Summarizing, when synthesizing a middleware architecture, mechanisms that detect idle state are incorporated within the resulting architecture. An idle state detection mechanism consists of a set of middleware components that monitor interaction between the application and the middleware. Each one of those components is a wrapper to the client-side proxy of either an application component, or a middleware service that is explicitly used by the application. The aforementioned set of wrappers needs to be coordinated so as to come up with a decision of whether the middleware is in an idle state, based on the monitoring information coming from the individual wrappers. This responsibility is assigned to the *coordinator* of the middleware exchange process. Another option is to introduce a separate coordinator for each set of wrappers that make up an idle state detection mechanism, or to allow the individual wrappers to interact. This, however, complicates the design and introduces an additional performance penalty.

Theories that describe properties of an idle state detector are based on the following pattern.

```
THEORY IdleState-Detection

include Exchangeable-Middleware-Style

value Pending_Property : PENDING
value Detector : C


AXIOM Report-Idle-State
∀X : XSituation |
   (∃C : C,
   σ : Σ |
      ⋄(IdleState(σ, X.MdwOld, Pending_Property)∧
      ⋄UpCall(X.MdwOld, C, Detector, X.reqXBEG, X.xid)) ⇔
      ReturnUpCall(X.MdwOld, C, Detector, X.reqXBEG, X.xid)

AXIOM Set-Pending-True-Condition
      % Define the conditions under which
      % Pending_Property(req, rid)
      % is assigned to true.

AXIOM Set-Pending-False-Condition
      % Define the conditions under which
      % Pending_Property(req, rid)
      % is assigned to false.

END
```

More specifically, the above theory relies on the `Exchangeable-Middleware-Style` theory. The theory defines the `Report-Idle-State` axiom, which states that if an idle state detector is notified about the beginning of an exchange situation, and an idle state is reached, it then reports back idle state. Moreover, the `Set-Pending-True-Condition`, `Set-Pending-False-Condition`, axioms describe the conditions under which a request is considered as being pending, and the conditions under which a request is no longer considered as being pending; concrete examples of such conditions are given in the case study section that follows.

Alternatively, if available and needed for reducing the application disruption, safe state detectors, specific to the particular exchange situation, can be installed at the time when this situation arises. This makes it possible to handle cases where waiting for the idle state causes a disruption (e.g. a lot of missed dead-lines for a real-time application) that can not be tolerated by the application. Detecting a safe state amounts to reasoning about whether it is possible to map the state of the old middleware into a state of the new middleware,

which is capable to satisfy requirements regarding requests that are currently pending. The state of the middleware changes as the application interacts with the middleware and as the architectural elements that make up the middleware interact with each other. Hence, detecting a safe state can be achieved by monitoring the interaction between the application and the middleware, and between the architectural elements that make up the middleware. In this case, the overall design of a mechanism that detects safe state is similar to the design of a mechanism that detects idle state. The safe state detector comprises a set of wrappers that wrap client-side proxies of the application components and client-side proxies of components that make up the overall middleware architecture. Another option for detecting safe state is by directly inspecting the state of the middleware. Let us note at this point that the ability to inspect the state of the middleware is generally promoted by reflective middleware infrastructures. In [14] for example, it is possible to easily inspect the state of the middleware through functionality exposed by the resource meta-model. In this case, a mechanism that detects safe state comprises components that access the reified state of the middleware.

As with the case of idle state detectors, theories that describe properties of a safe state detector are based on the following pattern.

```
THEORY SafeState-Detection

include Exchangeable-Middleware-Style

value Pending_Property : PENDING
value Detector : C


AXIOM Report-Safe-State
∀X : XSituation |
   (∃C : C,
   σ : Σ |
      ⏀(SafeState(σ, X.MdwOld, X.MdwNew, Pending_Property, X.Map)∧
      ⏀UpCall(X.MdwOld, C, Detector, X.reqXBEG, X.xid)) ⇔
      ReturnUpCall(X.MdwOld, C, Detector, X.reqXBEG, X.xid)

AXIOM Set-Pending-True-Condition
      % Define the conditions under which
      % Pending_Property(req, rid)
      % is assigned to true.

AXIOM Set-Pending-False-Condition
```

```
        % Define the conditions under which
        % Pending_Property(req, rid)
        % is assigned to false.

    END
```

Installing a new safe state detector just before the exchange is also an interesting issue and depends on whether the safe state detector monitors interaction, or inspects the middleware state. In order to detect the safe state correctly, the safe state detector must access information regarding requests that are currently pending. This is possible if the safe state detector is capable to inspect the state of the middleware. If, however, the safe state detector only monitors interaction, the middleware adaptation must take place in a moment when the corresponding idle state detector that is already in place reports that there is no request pending. Note that waiting for a detector to report idle state causes less inefficiencies than waiting for a middleware to reach an idle state. In the latter case, all idle state detectors should report idle state simultaneously.

In addition to the interfaces, required and provided by a safe state detector for monitoring interaction and inspecting the middleware state, the safe state detector provides an interface used by the coordinator for notifying the detector about the beginning and the end of a particular exchange situation.

IV.3.2 Selectively blocking threads of execution

The static part of the support implements the blocking process. As already said, the blocking process is not related with a particular exchange situation and, hence, there is no need for this part of the support to be exchangeable. Consequently, the static middleware part is incorporated within each middleware implementation during the synthesis process and remains unchanged for the lifetime of the application. In order to block threads issuing requests that are not necessary to drive middleware in a safe state, the static support must be able to monitor the interaction of application components, through the middleware. Consequently, the static support consists of a set of components each one of which wraps the client-side proxy of an application component. Except for the interfaces, typically provided (resp. required) by a wrapper, the wrappers that make up the static support provide an interface used by the coordinator to notify them about the beginning and the end of a particular exchange situation. When beginning the exchange process, the coordinator instructs the static support to install, if necessary, safe state detectors specific to the exchange situation. Changing parts of a middleware dynamically is more or less straightforward and strongly depends on the functionality provided by the underlying infrastructure, or the underlying operating system. For example, Java based middleware infrastructures like Ejb and JavaBeans can benefit from functionalities provided by the Java language, which allow to dynamically load and un-load executable code. In addition, Java provides a sophisticated mechanism that performs automatic garbage collection which

is useful for cleaning up leftovers, resulting from subsequent load and un-load actions. Moreover, most UNIX-like operating systems provide user libraries that allow to load and un-load fractions of executable source code. In the context of this thesis, we experimented with such a library, named DL, that comes along with SOLARIS. We used this library to dynamically exchange middleware implementations based on different CORBA compliant object request brokers.

The, so called, *reflective middleware infrastructures* are examples of flexible infrastructures that give a lot of functionality for changing dynamically parts of a middleware. Those infrastructures are based on the concept of reflection proposed by Smith in [81]. Based on those ideas, reflective middleware infrastructures support reification, i.e. they expose functionalities that allows changing the implementation of a middleware at runtime. Among the reflective middleware infrastructures we meet the reflective ORB proposed in [46], which reifies middleware proxies allowing them to be exchangeable at runtime. Similarly, the reflective ORB proposed in [80] reifies request marshaling, and invocation. Moreover, Flexinet [26], enables to dynamically tailor a basic middleware architecture by adding and removing layers that deal with security, marshaling etc. In the same spirit, Dynamic-TAO, [43], allows to dynamically load and unload code into a middleware implementation and enables inspection of the middleware state. The Open-ORB platform from Lancaster University [16] is even more flexible since it reifies bindings between components, resource information, and component interfaces and structure.

When completing the exchange process the coordinator instructs the static support to load the new middleware in place of the old one. In the context of this document, it is assumed that the update is done by unloading the old middleware implementation and loading the new middleware implementation. However, reflective middleware infrastructures like the ones previously mentioned provide functionalities that allow to do more modular changes. In [15] for example, it is possible to easily change parts of a middleware by accessing the compositional meta-model. Given that the functionality is available, it is preferable to perform modular changes. If, however, the new middleware architecture is much different than the old one there is no point in trying to modularize changes.

Finally, each one of the wrappers that constitute the static support is capable of collecting dependencies between threads of the application. Given those dependencies, and if an exchange situation arises, the wrapper selectively blocks threads, trying to issue requests through the middleware, that are not needed for driving middleware to a safe state. The theory that follows describes properties of components that constitute the static support. More specifically, the following theory is based on the `Exchangeable-Middleware-Style` theory and defines an axiom which states that during an exchange situation, only threads that are necessary for driving middleware in a safe state are allowed to issue requests through the middleware.

`THEORY Static Support`

```
include Exchangeable-Middleware-Style

AXIOM Blocking-Process
```
$\forall X : XSituation,$
$req : \mathcal{R},$
$rid : \mathcal{UID} \mid$
$\quad (\exists C, C', C'', C''' : \mathcal{C} \mid$
$\quad\quad Call(X.MdwOld, C, C', X.reqXBEG, X.xid) \wedge$
$\quad\quad \Diamond(Call(Conf, C'', C''', req, rid) \wedge$
$\quad\quad \Diamond Call(X.MdwOld, C, C', X.reqXEND, X.xid)) \Rightarrow$
$\quad \neg UnNecessary(Conf, CallTread(req, rid), X))$

```
END
```

IV.3.3 Coordinator

The coordinator is the last part of the support that needs to be incorporated within a middleware architecture for it to be exchangeable. The coordinator synchronizes the whole middleware exchange process. In particular, the coordinator notifies the static support for the beginning of the middleware exchange process and possibly instructs the static support to load safe state detectors specific to the particular exchange situation. Moreover, the coordinator notifies the safe state detectors about the beginning of the exchange process and asks them to report back the time when middleware reaches a safe state. Finally, at the time when all of the safe state detectors report that the middleware is in a safe state, the coordinator instructs the static middleware support to actually change the old middleware.

The following theory describes properties of the coordinator. More specifically, the `End-Exchange` states that the coordinator completes the middleware exchange process only after all of the safe state detectors report that the middleware has reached a safe state.

```
THEORY Coordinator

include Exchangeable-Middleware-Style

value
```
$StaticSupport : \{C : \mathcal{C}\} \neq \emptyset$
```
value
```
$Detectors : \{C : \mathcal{C}\} \neq \emptyset$
```
value
```
$Coordinator : \mathcal{C}$

```
AXIOM Begin-Exchange
```
$\forall X : XSituation \mid$
$\quad (\forall D, S : \mathcal{C} \mid$

$$D \in Detectors \wedge S \in StaticSupport \wedge$$
$$Call(X.MdwOld, Coordinator, S, reqXBEG, X.xid) \wedge$$
$$\Diamond Call(X.MdwOld, Coordinator, D, reqXBEG, X.xid))$$

AXIOM End-Exchange

$$\forall X : XSituation \mid$$
$$(\forall D, S : \mathcal{C} \mid$$
$$D \in Detectors \wedge S \in StaticSupport \wedge$$
$$Call(X.MdwOld, Coordinator, S, reqXEND, X.xid) \wedge$$
$$Call(X.MdwOld, Coordinator, D, reqXEND, X.xid)) \Rightarrow$$
$$\Diamond\!\!-(\forall D \mid$$
$$D \in Detectors \wedge$$
$$ReturnCall(X.MdwOld, Coordinator, D, reqXBEG, X.xid))$$

END

Given the design detailed in this section, and based on the properties of the instantiated elements that are incorporated within an exchangeable middleware architecture, it is possible to verify that the requirements for correct middleware exchange are satisfied.

IV.4 A Case Study

As in the previous chapters, a case study is presented so as to exemplify the systematic middleware exchange process. Starting again with the distributed file system application that requires a middleware, which provides transactions and reliable communication, this section first examines possible reasons that may lead to the need to adapt this middleware with another one. Then, the design of an exchangeable middleware architecture that satisfies the application requirements is presented. This design is based on the general design introduced in the previous section.

IV.4.1 Middleware adaptation

As discussed in Section II.3, the requirements of the distributed file system are those for reliable communication between clients and file servers, and transactions that guarantee the atomic and isolated execution of compound requests, issued from clients to file servers. Moreover, in the same section, it is mentioned that many variations of the traditional flat transaction model were proposed. The reason for that is because the traditional flat transaction model proved to be too strict and expensive for certain types of applications, which require more cooperation among the components involved in the execution of transactions.

Based on those remarks, suppose that the distributed file system application was initially designed for an enterprise that supports a small number of clients and was further

designed to host a relatively small number of file servers. Assuming that at some time the enterprise grows bigger, and that the flat transaction model imposes a significant performance cost, there is a need to change the current middleware with one that provides a more flexible transaction model like, for instance, join/split transactions. Moreover, the required change must take place with a minimal cost for the enterprise. This means that service provisioning should not be interrupted for long time, and even more, it should be safely interrupted so that the distributed file system does not exhibit a malfunctioning behavior to the clients. It is true that the former scalability problem could be solved with a more careful design of the distributed file system application, in which case the middleware exchange would not be necessary. Typically, however, the need to change a software system, originates from the fact that the architects, the designers, and the developers frequently make decisions based on predictions, estimations, or speculations which may finally prove to be incorrect. Since imperfection cannot be separated from the rest of the features that characterize the human nature, it is impossible to avoid the existence of incorrect decisions that need to be fixed with a minimal cost.

Getting back to our example, it is also possible that the need to exchange middleware originates from changes in the availability of software. More efficient and correct releases of infrastructures may lead to exchange a middleware used in an application with a better version. In our example, suppose that the distributed file system is built on top of the Orbix infrastructure, which was among the first Corba compliant Orb releases. However, the Orb evaluation by the CSD of Charles University, Prague [19], shows that more recent, Corba compliant object request brokers, like OmniOrb and VisiBroker perform much better than Orbix. Consequently, such a result triggers the need to safely exchange the existing transactional middleware with one that is built on top of one of those infrastructures, or on top of a new version of Orbix. Hence, the goal in our example scenario is to exchange the middleware architecture synthesized as described in Section III.4, with another one that may provide the same, or different properties compared to the old middleware.

IV.4.2 Middleware architecture design

Figure IV.2, gives the overall architecture of an exchangeable middleware that provides reliable communication and transactions. Note that the architectural elements depicted in this figure are composite. The reason for this is to simplify the figure by omitting unnecessary details.

The `Corba-Transactional-Middleware` element, represents an instance of the transactional middleware architecture, synthesized in Section III.4. In addition, to the previous architectural element, the exchangeable middleware architecture includes two idle state detection mechanisms. One of them is used to detect idle state with respect to the `Transactions` property, while the other is used to detect idle state with respect to the `Realistic-Reliable-Rpc` property. The former idle state detection mechanism consists of a wrapper, called `Transactions-IdleState`, that wraps the client-side proxy of the `Ots`

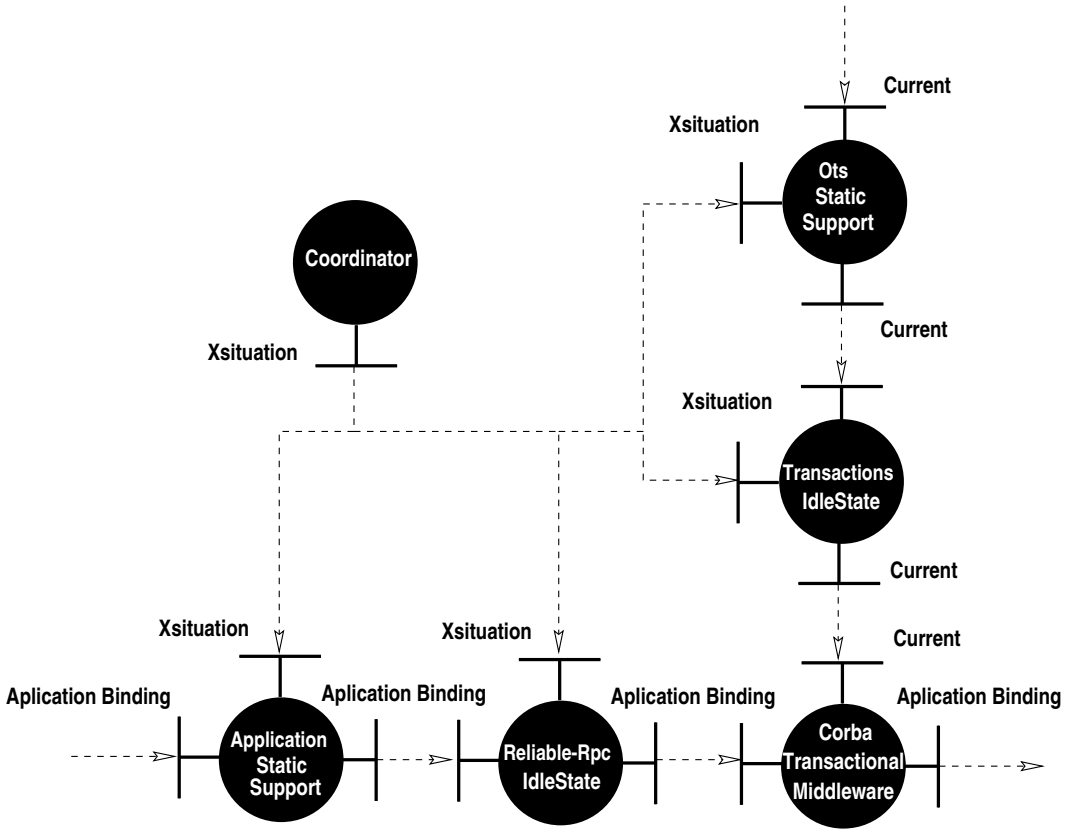

Figure IV.2 Case Study: An exchangeable middleware architecture

middleware service, which is explicitly used by the application components.

The following theory describes properties of the `Transactions-IdleState` component. In particular, the `Set-Pending-True-Condition`, `Set-Pending-False-Condition`, axioms describe the way the component monitors interaction that takes place through the middleware. With respect to the former axiom, every request req belonging to the context of a transaction t, is qualified as being pending at the time when the originator $t.O$ initiates t, by exporting the $t.reqB$ request. With respect to the latter axiom, all requests $req \in t.RXID$ are qualified as not pending at the time when the originator commits or aborts t.

```
THEORY Transactions-IdleState-Detection

include Transactional-Style
include Exchangeable-Middleware-Style

value Pending_Transactions : PENDING
value Detector : C
```

```
AXIOM Report-Idle-State
% As in Section IV.3.1

AXIOM Set-Pending-True-Condition
```
$$\forall t : \mathcal{T} \mid$$
$$Call(Conf, t.O, Mdw, t.reqB, t.tid) \Rightarrow$$
$$\forall req : \mathcal{R},$$
$$rid : \mathcal{UID} \mid$$
$$(req, rid) \in t.RXID \Rightarrow Pending_{Transactions}(req, rid) = true$$

```
AXIOM Set-Pending-False-Condition
```
$$\forall t : \mathcal{T} \mid$$
$$(Call(Conf, t.O, Mdw, t.reqC, t.tid) \vee Call(t.O, Mdw, t.reqC, t.tid)) \Rightarrow$$
$$\Diamond(\forall req : \mathcal{R},$$
$$rid : \mathcal{UID} \mid$$
$$(req, rid) \in t.RXID \Rightarrow Pending_{Transactions}(req, rid) = false)$$

```
END
```

The other idle state detection mechanism consists of a component, `Reliable-Rpc-IdleState`, that wraps the client-side proxy of the application. The following theory describes properties of the `Reliable-Rpc-IdleState` component. Similarly to the previous theory, the one that follows describes the way that the `Reliable-Rpc-IdleState` component monitors the interaction between the application, through the middleware.

```
THEORY Reliable-Rpc-IdleState-Detection

include Exchangeable-Middleware-Style

value
```
$Pending_{Reliable-Rpc} : \mathcal{PENDING}$
```
value
```
$Detector : \mathcal{C}$
```
value
```
$Conf : \mathcal{CONF}$

```
AXIOM Report-Idle-State
% As in Section IV.3.1

AXIOM Set-Pending-True-Condition
```
$$\forall C, C' : \mathcal{C},$$

$req : \mathcal{R},$
$rid : \mathcal{UID}$ |
$Call(Conf, C, C', req, rid) \Rightarrow$
$Pending_{Reliable-Rpc}(req, rid) = true$

`AXIOM Set-Pending-False-Condition`
$\forall C, C' : \mathcal{C},$
$req : \mathcal{R},$
$rid : \mathcal{UID}$ |
$(\exists req' : \mathcal{R}$ |
$UpCall(Conf, C', C, req', rid) \wedge failure(req'))\vee$
$(\nexists resp : \mathcal{R}$ |
$\Diamond ReturnUpCall(Conf, C, C', resp, rid)\wedge$
$UpCall(Conf, C, C', req, rid))\vee$
$(\exists resp : \mathcal{R}$ |
$ReturnCall(Conf, C, C', resp, rid)) \Rightarrow$
$Pending_{Reliable-Rpc}(req, rid) = false$

`END`

Furthermore, the exchangeable middleware architecture shown in Figure IV.2 comprises two components that constitute the static support. The `Application-Static-Support` component, monitors the interaction between application components through the middleware. The `Ots-Static-Support` component, monitors the interaction of the application with the `Ots` component. Finally, the exchangeable middleware architecture comprises a component, named `Coordinator`, that coordinates the exchange process.

IV.5 ADL Evaluation

As mentioned in Section II.1.4, an ADL must allow modeling the possible dynamic evolution of a configuration. Hence, it should provide means for specifying changes in component and connector instances that constitute a running configuration. Moreover, the architecture description language should come along with adequate tools that safely perform changes in a running configuration.

Among the ADLs examined in the context of this document, CONIC, DARWIN, DURRA, C2, and finally ASTER, view configurations dynamically. More specifically, CONIC provides language primitives that allow to describe the instantiation of components, and the re-binding of ports. Those primitives can be used to describe changes in an existing configuration. ADL specifications that describe changes in a running configuration are applied by a configuration manager that comes along with the ADL. Similarly, DARWIN provides language primitives allowing to describe the dynamic instantiation, removal and re-binding

of ports. A DARWIN specific configuration manager is used to apply changes into a running configuration. The basic language primitives that describe dynamic instantiation and port re-binding can be used by both the application components and the application administrator, as opposed to CONIC where the use of the language primitives by the application is not possible. In RAPIDE, dynamic configuration changes are modeled in terms of rules for the creation and connection of components specified by the architect. A creation rule specifies a poset, whose occurrence triggers the creation of a new component. Similarly, a connection rule describes a poset, whose occurrence results in setting a connection between components. The compilation tools that come along with the RAPIDE ADL make sure that connection and creation rules are properly applied at runtime. In the same spirit, in DURRA, runtime configuration changes are modeled in terms of reconfiguration rules. A reconfiguration rule describes a transition from one configuration to another configuration. Possible transitions are described at design time and they are applied at runtime by a set of configuration managers. Each manager out of this set is responsible for changing a particular cluster of component instances. In C2, configuration changes are modeled as sequences of primitive reconfiguration actions like component creation, removal, welding etc. The language provides corresponding primitives allowing to express such reconfiguration actions. Those actions are applied onto a running configuration with using a tool named ArchShell [55]. In ASTER, dynamic configuration management of CORBA applications is done by using an extended implementation of the CORBA LifeCycle service called DRS [11]. The DRS, thus, implements basic primitives of the LifeCycle service, which allow to create, destroy and move CORBA objects, and further guarantees that those actions are performed without violating RPC integrity. The support provided along with the ADL then allows to easily retrieve (based on given requirements for dynamic configuration management for a CORBA application), and integrate a CORBA application with the DRS service. At this point it must be noted that ASTER, in contrast to other ADLs that are based on a standard middleware infrastructure, does exploit standard services provided by the underlying middleware infrastructure towards the dynamic configuration management of an application. Nowadays, support for developing, deploying and maintaining an application is provided by middleware infrastructures. In the case of CORBA-compliant infrastructures, there exist common object services like the LifeCycle or the Trading service, just for managing the configuration of a running application. In EJB and DCOM, the facts are similar. In the former case, components life-cycle is managed by an EJB container. In the latter case, an MTS executive can be used for creating and destroying DCOM components. Consequently, an ADL and its tools should provide a foundation, that eases the design, development and maintenance of software. In other words, an ADL and its tools should provide a methodology for designing, developing and maintaining software. This methodology should enable exploiting the features of services provided by existing technologies.

However, ASTER so far, as most of the other ADLs, was only dealing with the dynamic configuration management of the application components. Excepting DCL, none of the ADLs examined confronts changes in the middleware connector that mediates the interactions among the application components. In DCL, both modules and protocols are

considered as being re-configurable architectural elements. However, the issue of consistent and non-disrupting reconfiguration of modules protocols is not examined. In this thesis, the author proposed an approach for exchanging middleware. Starting from the very beginning the author argued that supporting arbitrary middleware changes while preserving requirements like application consistency, reachability, and minimal application disruption, cannot be satisfied with using previously proposed methods for reconfiguration. What is needed for exchanging middleware while preserving the previous requirements is to do the exchange with using software that is specific to a particular exchange situation. Consequently, support for exchanging middleware is not general purpose and cannot be implemented as a general purpose ADL tool. However, the basic properties that should characterize the software used for exchanging middleware are general purpose. Those properties have been examined in this chapter and can be part of the proposed foundation for the systematic customization of middleware.

V Conclusion

The last chapter of this document presents a summary of the systematic customization of middleware and puts down value of the proposed approach. Moreover, this chapter briefly presents the current status of a prototype implementation and finally, points out open issues and future directions regarding the systematic customization of middleware.

V.1 Summary and Contribution

The development of software is much simplified when building on top of a middleware infrastructure. Typically, such an infrastructure deals with problems like heterogeneity, interoperability etc. in a way transparent to the application developer. Moreover, it supports services that provide solutions to frequently met problem like security, transactions, etc. However, building a middleware out of services provided by a middleware infrastructure is not a trivial task. This task, typically, comprises refining the requirements of an application into a concrete middleware architecture that satisfies them, selecting implementations of the appropriate services that make up the concrete architecture, assembling those implementations into a middleware, integrating the result within the application, and finally maintaining the middleware during the lifetime of the application.

The main objective of this thesis was to provide a systematic method for the customization of middleware that meets requirements of an application. The solution proposed for achieving this goal is based on the software architecture paradigm for developing software. More specifically, the basic idea is to use an architecture description language to precisely describe the structure and the properties of software, and to devise adequate tool support that helps the architect, the designer, and the developers to customize middleware based on the ADL description of a given application.

The proposed support for the systematic customization of middleware comprises a middleware repository that is used to store a history of previous refinement steps performed by the architect. Moreover, the proposed middleware repository contains implementations of concrete middleware architectures that resulted from the aforementioned refinement steps. The architect is provided with a systematic method for searching the refinement history for paths that may lead to concrete middleware architectures that refine the requirements

of the application, which is currently under development. The same method helps the designer to locate implementations that realize a concrete middleware architecture. The middleware repository is strongly inspired by the one previously proposed by ASTER and contributes with enhancing this idea. The basic enhancements suggested in this document are the following two. First, the middleware repository contains a full architectural description of the middleware architectures stored within it. Previously, the middleware repository included only descriptions of the properties provided by a middleware architecture, making it difficult to trace changes across different refinement levels regarding the way abstract middleware architectural elements decompose into more concrete middleware architectural elements. The second enhancement relates to the method adopted for navigating through the repository. Previously, the middleware repository was searched using only theorem proving technology. In this document, a hybrid search method is proposed in order to decrease the use of a theorem prover when searching for a middleware architecture that meets application requirements. The first step of the proposed search method comprises searching the repository for middleware architectures that meet requirements for a particular architectural style. This step is achieved without the use of a theorem prover. The second step starts from the set of middleware architectures resulted from the first step, and aims at locating middleware architectures whose properties satisfy the properties required by the application. this step is realized using a theorem prover.

The support for the systematic customization of middleware further includes a method for integrating implementations of concrete middleware architectures that can be easily reused within different applications. As discussed in previous chapters, architecture description languages that aim at easing the software development process, either come along with an ADL specific middleware infrastructure and tools that allow to integrate an application with a middleware that is based on this infrastructure (e.g. DURRA, C2), or provide tools, specific to well-known infrastructures like CORBA, DCOM, EJB etc., that allow to integrate an application with a middleware built on top of them (e.g. ASTER, DARWIN). The contribution of this thesis regarding those approaches is that the method proposed for integrating the implementation of a concrete middleware architecture is not specific to the infrastructure that the middleware relies on.

The support for the systematic customization of middleware, finally comprises a method for building middleware architectures that can adapt so as to reflect changes in the application requirements and the availability of middleware. Middleware adaptation must take place in a finite time and disrupt the application as less as possible. Furthermore, middleware adaptation must not break the application consistency. In the worst case where the application requirements and the availability of middleware change arbitrarily, an existing middleware architecture must be exchanged with a new one. From the architectural point of view, exchanging a middleware with another one is equivalent to exchanging connectors that mediate the interaction among components. Few of the ADLs examined in the context of this thesis provide support for changing application components but none deals with consistent changes performed on connectors. This thesis contributes with the definition of an architectural style that can be followed towards designing middleware architectures that

can be exchanged while introducing minimal application disruption and without breaking the application consistency.

Regarding the requirement for minimal application disruption, previous work in dynamic configuration management proposed to identify a minimal set of components affected by the change, and to, at most, disturb those components while the rest of the application continues its normal execution. However, when changing a middleware, the problem is that every application component that interacts with the middleware is affected by the change. Consequently, there is no point to try minimizing the set of components affected by the change, since the whole application is affected by the change. Given those remarks the main objective becomes to minimize the duration of the exchange. This is achieved using support that detects whether, or not it is safe to perform the exchange, which is specific to a particular exchange situation. In order to confront arbitrary changes in the application requirements and the availability of middleware, which can not be predicted, the support that detects whether it is safe to change the middleware must adapt regarding a particular exchange situation. In order to exchange a middleware within a finite time, the middleware architecture must include support that monitors the interaction between the application and the middleware and selectively blocks requests that are not needed to drive middleware in a state where it is safe to exchange it.

V.2 Current Status

According to the definition of refinement given in Section III.1.2, checking whether a middleware architecture refines another one requires proving that the properties provided by the latter are at least as strong as the ones provided by the former. Based on this fact and towards constructing the design repository described in Section III.2, the automatic and the interactive proving facilities of the STeP theorem prover have been tested. The overall result was that the automatic proving facilities work relatively well for simple proofs relying on the combination of few axioms. On the contrary, when getting to more complicated theorems whose proof requires the combination of a large number of axioms the use of the interactive prover can not be avoided. Moreover, the current version of STeP seems to have problems with sets and quantifiers. This led us in trying to avoid the use of such constructs in the description of properties that characterize the observable behavior of middleware. Due to the lack of time, no other theorem provers were tested, however, this is among the tasks scheduled for the near future.

A tool that integrates a middleware implementation within a given application has been developed. This tool accepts as input a process described using the simple language described in Section III.4, and a symbol table that contains architectural information specific to the application. Then, according to the process directives, the application-dependent parts of the middleware are automatically generated. Code generation takes place in two steps. During the first step, the development process is parsed resulting in a tree of process directives that should be followed to generate the application specific parts of the middle-

ware implementation. The second step, is a bottom up traversal of the tree, during which the process directives are expanded, depending on their type and based on the application specific ADL information.

Finally, regarding the systematic middleware exchange process, the basic ideas proposed in this document were applied towards exchanging middleware architectures implemented on top of different CORBA compliant middleware infrastructures.

V.3 Perspectives

At the very beginning of this thesis, it is assumed that an architect gets as input abstract requirements of an application and tries to refine them into a concrete middleware architecture that satisfies them. Then, this thesis proposes using a design repository that records previous refinement steps performed by the architect. This thesis further proposes a systematic method that helps the architect to browse this history towards locating refinement paths that can be reused for refining the requirements of the application that is currently under development. This approach, however, is kind of simplistic. As discussed in [40], the requirements of an application comprise a set of distinct and possibly unlike properties that should be provided by a middleware. For example, application requirements may be as diverse as dependability, security, transactions etc. Consequently, more than one expert is needed for refining the application requirements into a concrete middleware architecture. Each one of those experts concentrates on refining an architectural view of the middleware that reflects only properties belonging to his area of expertise. Based on those remarks, an ADL should provide support for coordinating the work performed by the different experts. More specifically, what is required is support that allows to compose different architectural views, and to check whether the composed architecture is consistent given some well-defined constraints.

An equally interesting issue that still relates to the refinement relation is to identify ways to systematically reason about specific changes that should be applied into a concrete middleware architecture starting from specific changes into the requirements of the application. This issue would improve the proposed method for changing a middleware due to a change in the application requirements. The proposed method assumes, so far, that the whole middleware architecture is exchanged with another one. Nevertheless, more fine-grained changes can be identified by the architect, or the designer. Performing, however, this task is rather complicated and a systematic method that simplifies it would be of a great help for the architect. A good starting point towards such a method could be the approach proposed in [59], where the authors define the distance between specifications of programs.

The application designer is provided with an implementation repository and a systematic method for locating and retrieving available implementations of elements that can be used to realize a concrete middleware architecture. Then, it is his work to evaluate the

available implementations regarding criteria like performance, scalability etc. Performing the latter task is not helped at all given the proposed ADL support for the systematic customization of middleware. Trying to assist the designer towards evaluating properties related to the implementation of primitive architectural elements is a challenging issue. In the case of performance evaluation, for instance, the designer could be provided with both theoretical and practical metrics like a framework for the analysis of algorithmic complexity and a framework for benchmarking middleware [19], respectively.

Bibliographie

[1] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE, 1994.

[3] R. J. Allen, D. Garlan, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, November 1998.

[4] K.R. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[5] M. C. Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures*. PhD thesis, University of Illinois, 1999.

[6] M. C. Astley and G. A. Agha. Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering*, pages 1–9. ACM-SIGSOFT, Nov 1998.

[7] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. DURRA: A Structure Description Language for Developing Distributed Applications. *Software Engineering Journal*, pages 83–94, March 1993.

[8] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[9] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[10] C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of the 5th European Symposium on Research in Computer Security*, pages 51–66. Spinger-Verlag, September 1998.

[11] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 35–42. IEEE, May 1998.

[12] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, M. Pichora, H.B. Sipma, and T.E. Uribe. *STeP The Stanford Temporal Prover Educational Release*. Stanford University, 1.4-a edition, July 1998.

[13] G. Blair, L. Blair, V. Issarny, P. Tůma, and A. Zarras. The role of software architecture in constraining middleware adaptation in component-based middleware platforms. In *Proceedings of MIDDLEWARE '00*. IFIP, 2000. Submitted for publication.

[14] G. Blair, F. Costa, G. Coulson, F. Delpiano, H. Duran, B. Dumant, F. Horn, N. Parlavantzas, and J-B. Stefani. The Design of a Resource-Aware Reflective Middleware Architecture. In *Proceedings of REFLECTION '99*, pages 115–134. ACM SIGPLAN, AITO, Springer-Verlag, July 1999.

[15] G. S. Blair, L. Blair, and J.-B. Stefani. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and* ISDN *Systems*, 29, 1997.

[16] G.S. Blair, G. Goulson, and M. Papathomas. An Architecture for the Next Generation Middleware. In *Proceedings of MIDDLEWARE '98*, pages 191–203. IFIP, September 1998.

[17] P. C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.

[18] R. P. Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

[19] DSR Group and MLC Systeme. Corba comparison project. Technical report, Charles University and MLC Systeme Gmbh, August 1999. http://nenya.ms.mff.cuni.cz/thegroup/.

[20] A. E. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[21] B. Fischer. Specification Based Browsing of Software Component Libraries. In *Proceedings of the 13th International Conference on Automated Software Engineering*. IEEE, October 1998.

[22] W. B. Frakes and P. B. Gandel. Representing Reusable Software. *Information and Software Technology*, 32(10), 1990.

[23] K.M. Goudarzi and J. Kramer. Maintaining Node Consistency in the Face of Dynamic Change. In *3rd International Conference on Configurable Distributed Systems*, pages 62–69. IEEE, May 1996.

[24] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference in VLDB*, pages 144–154, September 1981.

[25] S. Hauptmann and J. Wasel. On-line Maintenance with On-the-Fly Software Replacement. In *3rd International Conference on Configurable Distributed Systems*, pages 70–80. IEEE, May 1996.

[26] R. Hayton, A. Herbert, and D. Donaldson. Flexinet - A Flexible Component Oriented Middleware System. In *Proceedings of the 8th ACM-SIGOPS European Workshop*. ACM, Sept 1998.

[27] S. Henninger. Using Iterative Refinement to Find Reusable Software. *IEEE Software*, 11(5):48–59, September 1994.

[28] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, April 1997.

[29] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communication of the ACM*, 12(10):576–583, October 1969.

[30] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[31] C.A.R. Hoare. Unifying theories: a personal statement. *ACM Computing Surveys*, 28(4), December 1996.

[32] IONA. *Orbix Advanced Programmers Manual*. IONA Technologies Ltd, 2 edition, October 1997. http://www.iona.com/.

[33] ISO/IEC. Reference Model for Open Distributed Processing Part 1. Overview. Technical Report 10746-1, ISO/IEC, 1995.

[34] ISO/IEC. Reference model for open distributed processing part 3. architecture. Technical Report 10746-3, ISO/IEC, 1995.

[35] V. Issarny. Configuration-Based Programming Systems. In *Proceedings of SOFSEM'97: Theory and Practice of Informatics*, pages 183–200. LNCS 1338, November 1997.

[36] V. Issarny, L. Bellisard, M. Riveill, and A. Zarras. *Recent Advances in Distributed Systems*, volume 1752, chapter Component-Based Programming of Distributed Systems. Springer-Verlag, 2000.

[37] V. Issarny and C. Bidan. Aster: A CORBA-based software interconnection system supporting distributed system customization. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, 1996.

[38] V. Issarny and C. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the Sixteenth IEEE International Conference on Distributed Computing Systems*, 1996.

[39] V. Issarny, C. Bidan, and T. Saridakis. Achieving middleware customization in a configuration-based development environment: Experience with the Aster prototype. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 207–214. IEEE, 1998.

[40] V. Issarny, T. Saridakis, and A. Zarras. Multi-View Description of Software Architectures. In *Proceedings of the 3rd International Software Architecture Workshop*, pages 81–84, November 1998.

[41] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–89. IEEE, 1994.

[42] J. J. Kenney. *Executable Formal Models of Distributed Transaction Systems based on Event Processing*. PhD thesis, Stanford University, 1996.

[43] F. Kon and R.H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems (COOTS'99)*. USENIX, May 1999.

[44] J. Kramer and J. Magee. The Evolving Philosophers Problem. *IEEE Transactions on Software Engineering*, 15(1):1293–1306, November 1990.

[45] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[46] T. Ledoux. Implementing Proxy Objects in Reflective ORBs. In *Proceedings of ECOOP '97 Workshop on CORBA Implementation, Use and Evaluation*. ECOOP, 1997.

[47] S.L. Lo and D. Riddoch. *omniORB2 User's Guide*. AT&T Laboratories Cambridge, 2.7.1 edition, February 1999. http://www.uk.research.att.com/omniORB/.

[48] D. C. Luckham, J. Kenney, L. Augustin, J. Verra, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[49] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept 1995.

[50] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. In *Proceedings of the 1st International Conference on Configurable Distributed Systems*, March 1992.

[51] J. Magee, J. Kramer, and D. Giannakopoulou. Behavior analysis of software architectures. In *Proceedings of the 1st Working Conference on Software Architecture*, pages 35–49. IFIP, February 1999.

[52] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in CONIC. *IEEE Transactions on Software Engineering*, 16(5):663–675, June 1989.

[53] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[54] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *Proceedings of the 2nd International Software Architecture Workshop (ISAW-2)*, pages 24–27, October 1996.

[55] N. Medvidovic, P. Oreizy, and R.N. Taylor. Reuse of Off-the-Self Components in C2-Style Architectures. In *Proceedings of the International Conference on Software Engineering*, pages 692–700, May 1997.

[56] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Joint European Software Engineering - ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 60–76, September 1997.

[57] Microsoft. Component Services : A Technical Overview. Technical report, Microsoft Corporation, 1998. http://www.microsoft.com/com/wpaper/.

[58] Microsoft. DCOM: A Technical Overview. Technical report, Microsoft Corporation, 1998. http://www.microsoft.com/ntserver/appservice/.

[59] R. Mili, R. Mittermeir, and A. Mili. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1997.

[60] R. Milner. *A Calculus of Communicating Systems*. Cambridge University Press, 1980.

[61] R. Milner. *Communicating and Mobile Systems: the pi-calculus*. Springer-Verlag, 1999.

[62] N.H. Minsky, V. Ungureanu, and W. Wang. Building Reconfiguration Primitives into the Law of a System. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 89–97. IEEE, May 1996.

[63] F. Mish, editor. *The Merriam Webster Dictionary*. Merriam Webster Inc., 10th edition, July 1994.

[64] M. Moriconi, X. Qian, and A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[65] E. Moss. Nested Transactions and Reliable Distributed Computing. In *Proceedings of the International Conference on Reliability in Distributed Software and Database Systems*, pages 33–39. IEEE, July 1982.

[66] M. Nodine, S. Ramaswamy, and S. Zdonic. A Cooperative Transaction Model for Design Databases. In *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[67] OMG. UML Notation Guide. Technical Report 1.1, OMG Document, 1997.

[68] OMG. CORBAservices : Common Object Services Specification. Technical report, OMG Document, 1998.

[69] OMG. The Common Object Request Broker: Architecture and Specification – Revision 2.2. Technical report, OMG Document, 1998.

[70] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[71] F. Plasil and M. Stal. An Architectural View of Distributed Objects an Components in CORBA, Java RMI, and COM/DCOM. *Software Concepts and Tools*, 19(1), 1998.

[72] C. Pu, G. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference in VLDB*, pages 26–37, 1988.

[73] J. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1), 1994.

[74] K. Rommer, A. Puder, and F. Pilhofer. *MICO is CORBA. An Open Source CORBA 2.3 Implementation*. MICO, Sept 1999. http://www.mico.org/.

[75] T. Saridakis, C. Bidan, and V. Issarny. A Programming System for the Development of TINA Services. In *Proceedings of the International Conference on Open Distributed Processing*, 1997.

[76] T. Saridakis and V. Issarny. Developing Dependable Systems Using Software Architecture. In *Proceedings of the 1st Working Conference on Software Architecture*, pages 83–104. IFIP, February 1999.

[77] J. Schumman and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of the 12th International Conference on Automated Software Engineering*, pages 246–254. IEEE, November 1997.

[78] M. Shaw, R. Deline, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[79] S.K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of the Arjuna Distributed Programming System . *IEEE Software*, pages 66–73, January 1991.

[80] A. Singhai, A. Sane, and R. Campell. Reflective ORBs: Supporting Robust, Time Critical Distribution. In *Proceedings of ECOOP '97 Workshop on Reflective Real-Time Object Oriented Programming and Systems*. ECOOP, 1997.

[81] B. C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, 1982.

[82] D. C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois, 1994.

[83] SunMicroSystems. JavaBeans - Revision 1.01. Technical report, Sun MicroSystems, 1996. http://java.sun.co/beans.

[84] SunMicroSystems. Java Remote Method Invocation Specification - Revision 1.1. Technical report, Sun MicroSystems, October 1997. http://java.sun.com/products/JDK/.

[85] SunMicroSystems. Enterprise JavaBeans Technology. Technical report, Sun MicroSystems, 1998. http://java.sun.com/products/ejb/.

[86] P. Tůma, V. Issarny, and A. Zarras. Towards systematic synthesis of reflective middleware. In *Proceedings of REFLECTION '99*, pages 144–147. ACM SIGPLAN, AITO, Springer-Verlag, July 1999.

[87] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component and Message Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, July 1996.

[88] TINA-C. Overall Concepts and Principles of TINA. Technical report, TINA-C Document, February 1995.

[89] I. Warren and I. Sommerville. A Model for Dynamic Configuration which Preserves Application Integrity. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 81–88. IEEE, May 1996.

[90] S.M. Wheater, S.K. Shrivastava, and F. Ranno. A CORBA Compliant Transactional Workflow System for Internet Applications. In *Proceedings of MIDDLEWARE'98*, pages 3–18. IFIP, September 1998.

[91] A. M. Zaremski and J.M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

[92] A. Zarras and V. Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Proceedings of MIDDLEWARE'98*, pages 257–272. IFIP, Sept 1998.

[93] A. Zarras and V. Issarny. Imposing Transactional Properties on Distributed Software Architectures. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 25–32. ACM, Sept 1998.

[94] A. Zarras, P. Tůma, and V. Issarny. Using Software Architecture Description for the Systematic Customization of Middleware. *IEEE Concurrency*, 1999. Submitted for publication.

Résumé

Le traitement de problèmes rencontrés dans la construction de différentes familles d'applications a donné lieu à la définition et à la standardisation d'une couche logicielle qui se situe entre l'application et le système d'exploitation sous-jacent. Cette couche est connue sous le nom de *middleware*, et fournit des solutions réutilisables aux problèmes récurrents dans la construction de logiciels complexes, comme l'hétérogénéité, l'interopérabilité, la sécurité, la tolérance aux fautes ou encore l'exécution de transactions. Un *middleware* est typiquement construit à partir de services fournis par une infrastructure. Des exemples connus de telles infrastructures sont celles conformes au standard Corba, Dcom ou encore Ejb. L'implémentation d'un logiciel s'appuyant sur des solutions réutilisables, fournies par les infrastructures *middleware*, simplifie de manière évidente le processus de développement du logiciel. Les développeurs de logiciels se trouvent affranchis de l'implémentation de protocoles de gestion des communications, de la sécurité ou encore de modèles transactionnels. Étant donnée une infrastructure *middleware*, le travail du développeur quant à la mise en œuvre d'un système d'exécution (ou *middleware*) pour une application donnée est la combinaison de services *middleware* disponibles de telle sorte que le système résultant satisfasse les exigences de l'application. L'effort à fournir porte donc sur la conception d'architectures *middleware* qui satisfont les exigences particulières des applications. Notre travail de thèse a porté sur l'exploitation de la notion d'architecture logicielle pour la construction systématique de *middleware*, adaptés aux applications. Plus précisément, nous proposons l'utilisation d'un langage de description d'architectures et d'un ensemble d'outils associés pour systématiser : la conception d'une architecture *middleware* qui réponde aux besoins d'une application donnée, l'intégration de cette architecture au sein de l'application, et la maintenance de cette architecture au regard des évolutions relatives aux exigences de l'application ou à la disponibilité des services *middleware*.

La solution proposée à la construction systématique de *middleware* adaptés aux besoins des applications comprend un système de stockage qui garde trace de l'historique des conceptions d'architectures *middleware*. Ce système de stockage contient en outre les implémentations d'architectures concrètes qui résultent des étapes de conceptions mémorisées. Nous introduisons par ailleurs une méthode pour la localisation systématique de chemins de conceptions, *i.e.* séquences d'étapes de conceptions, qui conduisent éventuellement à des architectures concrètes de *middleware*, satisfaisant les exigences d'une application donnée. Cette facilité est complétée par une méthode de développement d'implémentations d'architectures concrètes de *middleware*, qui peuvent être aisément intégrées au sein d'une application. Enfin, nous donnons une solution à l'adaptation dynamique d'un *middleware* construit suivant notre méthode, qui préserve l'exécution cohérente de l'application s'appuyant sur celui-ci. Une telle fonctionnalité est essentielle pour la maintenance d'une architecture *middleware* au regard de l'évolution des exigences de l'application et de l'infrastructure *middleware*.